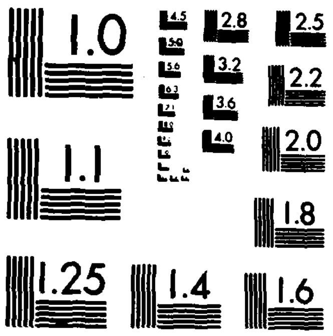

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

March 1986

Report No. STAN-CS-86-1125
Also numbered KSL-86-22

(2)

The CAOS System

by

Eric Schoen

Department of Computer Science

Stanford University
Stanford, CA 94305

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A174 178	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The CAOS System		5. TYPE OF REPORT & PERIOD COVERED technical
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-86-1126
7. AUTHOR(s) Eric Schoen		8. CONTRACT OR GRANT NUMBER(s) F30602-85-C-0012
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford Knowledge Systems Laboratory 701 Welch Road Bldg C Stanford University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE March 1986
		13. NUMBER OF PAGES 69
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See next page

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Abstract

The CAOS system is a framework designed to facilitate the development of highly concurrent real-time signal interpretation applications. It explores the potential of multiprocessor architectures to improve the performance of expert systems in the domain of signal interpretation.

CAOS is implemented in Lisp on a (simulated) collection of processor-memory sites, linked by a high-speed communications subsystem. The "virtual machine" on which it depends provides remote evaluation and packet-based message exchange between processes, using virtual circuits known as *streams*. To this presentation layer, CAOS adds (1) a flexible process scheduler, and (2) an object-centered notion of *agents*, dynamically-instantiable entities which model interpreted signal features.

This report documents the principal ideas, programming model, and implementation of CAOS. A model of real-time signal interpretation, based on replicated "abstraction" pipelines, is presented. For some applications, this model offers a means by large numbers of processors may be utilized without introducing synchronization-necessitated software bottlenecks.

The report concludes with a description of the performance of a large CAOS application over various sizes of multiprocessor configurations. Lessons about problem decomposition grain size, global problem solving control strategy, and appropriate services provided to CAOS by the underlying architecture are discussed.

Knowledge Systems Laboratory
Report No. KSL-86-22

March 1986

The CAOS System

Eric Schoen

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

The CAOS system is a framework designed to facilitate the development of highly concurrent real-time signal interpretation applications. It explores the potential of multiprocessor architectures to improve the performance of expert systems in the domain of signal interpretation.

CAOS is implemented in Lisp on a (simulated) collection of processor-memory sites, linked by a high-speed communications subsystem. The "virtual machine" on which it depends provides remote evaluation and packet-based message exchange between processes, using virtual circuits known as *streams*. To this presentation layer, CAOS adds (1) a flexible process scheduler, and (2) an object-centered notion of *agents*, dynamically-instantiable entities which model interpreted signal features.

This report documents the principal ideas, programming model, and implementation of CAOS. A model of real-time signal interpretation, based on replicated "abstraction" pipelines, is presented. For some applications, this model offers a means by large numbers of processors may be utilized without introducing synchronization-necessitated software bottlenecks.

The report concludes with a description of the performance of a large CAOS application over various sizes of multiprocessor configurations. Lessons about problem decomposition grain size, global problem solving control strategy, and appropriate services provided to CAOS by the underlying architecture are discussed.

Contents

Chapter 1

Introduction and Overview

This report documents the CAOS system, a portion of a recent experiment investigating the potential of highly concurrent computing architectures to enhance the performance of expert systems. The experiment focuses on the migration of a portion of an existing expert system application from a sequential uniprocessor environment to a parallel multiprocessor environment.[1]

The application, called ELINT, is a portion of a multi-sensor information fusion system, and was written originally in AGE[2], an expert system development tool based on the blackboard paradigm. For the purposes of this experiment, ELINT was reimplemented in CAOS, an experimental concurrent blackboard framework based on the explicit exchange of messages between blackboard agents.

CAOS, in turn, relies on services provided by the underlying machine environment. In the present set of experiments, the environment is a simulation of a concurrent architecture, called CARE [5]. CARE simulates a square grid of processing nodes, each containing a Lisp evaluator, private memory, and a communications subsystem; message-passing is the only means of interprocessor communication.

CAOS is principally an operating system, controlling the creation, initialization, and execution of independent computing tasks in response to messages received from other tasks. Figure 1.1 illustrates the relationship between the various software components of the experiment.

The following chapter briefly describes the salient features of the CARE environment. Chapter 3 discusses the ideas behind the CAOS framework. Chapter 4 summarizes the CAOS programming environment, and Chapter 5 describes its implementation. The final chapter details the results of our experiments. Finally, Appendix A contains a simple CAOS example, and Appendix B presents a detailed, low-level look at the implementation of CAOS.

[1]This research was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Eric Schoen was supported by a fellowship from NL Industries.

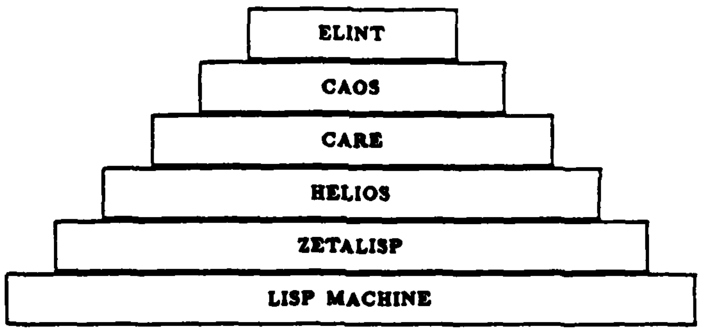

Figure 1.1: The relationship between ELINT, CARE, and CAOS

Chapter 2

An Overview of CARE

CARE is a highly-parameterized and well-instrumented multiprocessor simulation testbed, designed to aid research in alternative parallel architectures. It runs executes within Helios, a hierarchical, event-driven simulator which has been described elsewhere [3].

A typical CARE architecture is a grid of processing sites, interconnected by a dedicated communications network. For example, the research discussed in this paper was performed on square arrays of hexagonally connected processors (*e.g.*, each processor is connected to six of its eight nearest neighbors, excluding processors at the edges of the grid).

Each processing site consists of an *evaluator*, a general-purpose processor/memory pair, and an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator. Application-level computations take place in the evaluator, a component which is treated as a "black box" Lisp processor. No portion of its interior is simulated; the host Lisp machine serves as the evaluator in each processing site. The operator performs two duties. As a communications processor, it is responsible for routing messages between processing sites. As a scheduling processor, it queues application-level processes for execution in the evaluator (we discuss the scheduling mechanism in greater detail below). The operator is simulated and instrumented in great detail.

CARE allows a number of parameters of the processor grid to be adjusted. Among these parameters are: the speed of the evaluator, the speed of the communications network, and the speed of the process-switching mechanism. By altering these parameters, a single processor grid specification can be made to simulate a wide variety of actual multiprocessor architectures. For example, we can experiment with the optimal level-of-granularity of problem decomposition by varying the speed of both process-switching and communications.

Finally, CARE provides detailed displays of such information as evaluator, operator, and communication network utilization, and process scheduling latencies. This instrumentation package informs developers of CARE applications of how efficiently their systems make use of the simulated hardware.

2.1 The CARE Programming Model

CARE programs are made up of processes which communicate by exchanging messages. Messages flow across *streams*, virtual circuits maintained by CARE. The following services are used by CAOS:

New Process: Creates a new process on a specified site, running a specified top-level function. A new stream is returned, enabling the "parent" of the process to communicate with its "child." Pointers to the stream may be exchanged freely with other known processes on other sites.

New Stream: Creates a new stream whose target is the creating process.

Post Packet: Sends a message across a specified stream to a remote process.

Accept Packet: Returns the next message waiting on a specified stream. If no message is waiting when this operation is invoked, the invoking process is suspended and moved into the operator to await the arrival of a message.

Memory in each processing site is private. Ordinarily, intra-memory pointers may not be exchanged with processes in other sites. However, any pointer may be encapsulated in a *remote-address*, and may then be included in the contents of a message between sites. A remote address does not permit direct manipulation of remote structures; instead, it allows a process in one site to produce a local copy of a structure in another site.

Scheduling on a CARE node is entirely cooperative, and is based on message-passing. The message exchange primitives `post-packet` and `accept-packet` form the basis of process scheduling. A process wishing to block (yield control of the evaluator) does so by calling `accept-packet` to wait for a packet to arrive on a stream. The application program's scheduler awakens the process by calling `post-packet` to send a packet to the stream. The process is placed on the queue of processes waiting for the evaluator, and eventually regains control. The CAOS scheduler, which we describe in Section 5.3, is implemented in terms of this paradigm.

Chapter 3

The CAOS Framework

CAOS is a framework which supports the execution of multi-processor expert systems. Its design is predicated on the belief that future parallel architectures will emphasize limited communication between processors rather than uniformly-shared memory. We expected such an architecture would favor coarse-grained problem decomposition, with little or no synchronization between processors. CAOS is intended for use in real-time data interpretation applications, such as continuous speech recognition, passive radar and sonar interpretation, etc [7,11].

A CAOS application consists of a collection of communicating *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Furthermore, an arbitrary number of processes may be active at any one time in a single agent.

Whereas the uniprocessor blackboard paradigm usually implies pattern-directed, demon-triggered knowledge source activation, CAOS requires explicit messaging between agents; the costs of automatically communicating changes in the blackboard state, as required by the traditional blackboard mechanism, could be prohibitively expensive in the distributed-memory multiprocessor environment. Thus, CAOS is designed to express parallelism at a very coarse grain-size, at the level of knowledge source invocation in a traditional uniprocessor blackboard system. It supports no mechanism for finer-grained concurrency, such as within the execution of agent processes, but neither does it rule it out. For example, we could easily imagine the *methods* which implement the messages being written in QLisp [8], a concurrent dialect of Common Lisp.

3.1 The Structure of CAOS Applications

A CAOS application is structured to achieve high degrees of concurrency in two principal manners: *pipelining* and *replication*. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system; replication provides means by which the interpretation system can cope with arbitrarily high data rates.

3.1.1 Pipelining

Pipelining is a common means of parallelizing tasks through a decomposition into a linear sequence of independent stages. Each stage is assigned to a separate processing unit, which receives the output from the previous stage and provides input to the next stage. Optimally, when the pipeline reaches a steady-state, each of its processors is busy performing its assigned stage of the overall task.

CAOS promotes the use of pipelines to partition an interpretation task into a sequence of interpretation stages, where each stage of the interpretation is performed by a separate agent. As data enters one agent in the pipeline, it is processed, and the results are sent to the next agent. The data input to each successive stage represents a higher level of abstraction.

Advantages of Pipelining

Sequential decomposition of a large task is frequently very natural. Structures as disparate as manufacturing assembly lines and the arithmetic processors of high-speed computing systems are frequently based on this paradigm.

Pipelining provides a mechanism whereby concurrency is obtained without duplication of mechanism (that is, machinery, processing hardware, knowledge, etc). In an optimal pipeline of n processing elements, element 1 is performing work on task $t+n-1$ when element 2 is working on task $t+n-2$, and so on, such that element n is working on task t. As a result, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

In the case of CAOS applications, the individual agents which compose an interpretation "pipeline" are themselves simple, but the overall combination of agents may be quite complex.

Disadvantages of Pipelining

Unfortunately, it is often the case that a task cannot be decomposed into a simple linear sequence of subtasks. Some stage of the sequence may depend not only on the results of its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (*e.g.*, in feedback loops). An equally disadvantageous decomposition is one in which some of the processing stages take substantially more time than others. The effect of either of these conditions is to cause the pipeline to be used less efficiently. Both these conditions may cause some processing stages to be busier than others; in the worst case, some stages may be so busy that other stages receive no work at all. As a result, the n-element pipeline achieves less than an n-times increase in throughput. We discuss a possible remedy for this situation in the following section.

3.1.2 Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from individual processing elements to entire pipelines, is a candidate for replication. Consider a task which must be performed on average in time t, and a processing structure which is able to perform the task in time T, where $T > t$. If this task were actually a single stage in a larger pipeline, this stage would then be a bottleneck in the throughput of the pipeline. However, if the single processing structure which performed the task were replaced by

T/t copies of the same processing structure, the effective time to perform the task would approach t, as required.

Advantages of Replication

The advantages of replicating processing structure to improve throughput should be clear; n times the throughput of a single processing structure is achieved with n times the mechanism. Replication is more costly than pipelining, but it apparently avoids problems associated with developing a pipelined decomposition of a task.

Disadvantages of Replication

Our works leads us to believe that such replicated computing structures are feasible, but not without drawbacks. Just as performance gains in pipelines are impacted by inter-stage dependencies, performance gains in replicated structures are impacted by inter-structure dependencies.

Consider a system composed of a number of copies of a single pipeline. Further, assume the actions of a particular stage in the pipeline affects each copy of itself in the other pipelines. In an expert system, for example, a number of independent pieces of evidence may cause the system to draw the same conclusion; the system designer may require that when a conclusion is arrived at independently by different means, some measure of confidence in the conclusion is increased accordingly. If the inference mechanism which produces these conclusions is realized as concurrently-operating copies of a single inference engine, the individual inference engines will have to communicate between themselves to avoid producing multiple copies of the same conclusions. A stringent consistency requirement between copies of a processing structure decreases the throughput of the entire system, since a portion of the system's work is dedicated to inter-system communication.

3.2 An Example

We close this chapter by describing the organization of ELINT, illustrating the benefits and drawbacks of the CAOS framework applied to this problem. ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. Its goal is to correlate a large number of radar observations into a smaller number of individual signal emitters, and then to correlate those emitters into a yet smaller number of clusters of emitters. ELINT is meant to operate in real time; emitters and clusters appear and disappear during the lifetime of an ELINT run. The basic flow of information in ELINT is through a pipeline of the various agent types, which we now describe in detail.

Observation Reader

The observation reader is an artificat of the simulation environment in which ELINT runs. Its purpose is to feed radar observations into the system. The reader is driven off a clock; at each tick (1 ELINT "time unit"), it supplies all observations for the associated time interval to the proper observation handlers. This behavior is similar to that of a radar collection site in an actual ELINT setting.

Observation Handler

The observation handlers accept radar observations from associated radar collection sites (in the simulated system, the observations come from the observation reader agent). There may be a large number of observation handlers associated with each collection site. The collection site chooses to which of its many observation handlers to pass an observation, based on some scheduling criteria such as random choice or round-robin.

Each observation contains an externally-assigned number to distinguish the source of the observation from other known sources (the observation id is usually, but not always, correct). In addition, each observation contains information about the observed radar signal, such as its quality, strength, line-of-bearing, and operating mode. The observation does *not* contain information regarding the source's speed, flight path, and distance; ELINT will attempt to determine this information as it monitors the behavior of each source over time.

When an observation handler receives an observation, it checks the observation's id to see if it already knows about the emitter. If it does, it passes the observation to the appropriate emitter agent which represents the observation's source. If the observation handler does not know about the emitter, it asks an emitter manager to create a new emitter agent, and then passes the observation to that new agent.

Emitter Manager

There may be many emitter managers in the system. An emitter manager's task is to accept requests to create emitters with specified id numbers. If there is no such emitter in existence when the request is received, the manager will create one and return its "address" to the requesting observation handler. If there is such an emitter in existence when the request is received, the manager will simply return its address to the requestor. This situation arises when one observation handler requests an emitter than another observation handler had previously requested.

The reason for the emitter manager's existence is to reduce the amount of inter-pipeline dependency with respect to the creation of emitters. When ELINT creates an emitter, it is similar to a typical expert system's drawing a conclusion about some evidence; as discussed above, ELINT must create its emitters in such a way that the individual observation handlers do not end up each creating copies of the same emitter. Consider the following strategies the observation handlers could use to create new emitters:

1. The handlers could create the emitters themselves immediately. Since the collection site may pass observations with the same id to each observation handler, it is possible for each observation handler to create its own copy of the same emitter. We reject this method.

2. The handlers could create the emitters themselves, but inform the other handlers that they've done this. This scheme breaks down when two handlers try simultaneously to create the same emitter.

3. The handlers could rely on a single emitter manager agent to create all emitters. While this approach is safe from a consistency standpoint, it is likely to be impractical, as the single emitter manager could become a bottleneck in the interpretation.

4. The handlers could send requests to one of many emitter managers, chosen by some arbitrary method. This idea is nearly correct, but does not rule out the possibility of two emitter managers each receiving creation requests for the same emitter.

5. The handlers could send requests to one of many emitter managers, chosen through some algorithm which is invariant with respect to the observation id. This is in fact the algorithm in use in ELINT. The algorithm for choosing which emitter manager to use is based on a many-to-one mapping of observation id's to emitter managers.[1]

Emitters

Emitters hold some state and history regarding observations of the sources they represent. As each new observation is received, it is added to a list of new observations. On a regular basis, the list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine its heading, speed, and location. The first time it is able to determine this information, it asks a cluster manager to either *match* the emitter to an old cluster or *create* a new cluster to hold the single emitter. Subsequently, it sends an update message to the cluster to which it belongs, indicating its current course, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible*, *probable*, and *positive*). If new observations are received often enough, the emitter will increase its confidence level until it reaches *positive*. If an observation is not received in the expected time interval, the emitter lowers its confidence by one step. If the confidence falls below *possible*, the emitter "deletes" itself, informing its manager, and any cluster to which it is attached.

Cluster Managers

The cluster managers play much the same role in the creation of cluster agents as the emitter managers play in the creation of emitters. However, it is not possible to compute an invariant to be used as a many-to-one mapping between emitters. If ELINT were to employ multiple cluster managers, the best strategy for choosing which of the many managers would still result in the possible creation of multiple instances of the "same" cluster. Thus, we have chosen to run ELINT with a single cluster manager. Fortunately, cluster creation is a rare event, and the single cluster manager has never been a processing bottleneck.

As indicated above, requests from emitters to create clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading. However, the cluster manager does not itself perform this matching operation. Although it knows about the existence of each cluster it has created, it does not know if the cluster has changed course, speed, and/or direction since it was originally created. Thus, the cluster manager asks each of its clusters to perform a match.

If either none of the clusters responds with a positive match, a new cluster is created for the emitter; if one cluster responds positively, the emitter is added to the cluster, and is so informed of this fact; if more than one cluster responds positively, an error (or a mid-air collision) must have occured.

[1] The algorithm computes the observation id modulo the number of emitter managers, and maps that number to a particular manager.

Clusters

The radar emissions of clusters of emitters often indicates the actual behavior of the cluster. Cluster agents, therefore, apply heuristics about radar signals to determine whether the behaviors of the clusters they represent are threatening or not. This information, along with the course parameters of each radar source, is the "output" of the ELINT system. A cluster will delete itself if all constituent emitters have been deleted.

Chapter 4

Programming in the CAOS Framework

CAOS is package of functions on top of Lisp. These functions are partitioned into three major classes:

- Those which declare agents.
- Those which initialize agents.
- Those which support communication between agents.

We now describe the CAOS operators for each of these classes.

4.1 Declaration of agents

Agents are declared within an inheritance network. Each agent inherits the characteristics of its (multiple) parents. The simplest agent, **vanilla-agent**, contains the minimal characteristics required of a functional CAOS agent. All other CAOS agents reference **vanilla-agent** either directly or indirectly. Another predeclared agent, **process-agenda-agent**, is built on top of **vanilla-agent**, and contains a priority mechanism for scheduling the execution of messages.

Application agents are declared by augmenting the following characteristics of the base or other ancestral agents:

Local Variables: An agent may refer freely to any variable declared local. In addition, each local variable may be declared with an initial value.

Messages: The only messages to which an agent may respond are those declared in this table. This simplifies the task of a resource allocator, which must load application code onto each CARE site.

```
(defagent agent-name (parent_1 ··· parent_n)
  (localvars variable_1 ··· variable_n)
  (messages message_1 ··· message_n)
  (symbolically-referenced-agents agent_1 ··· agent_n))
```

Figure 4.1: The basic form of defagent

Symbolically Referenced Agents: Some agents exist throughout a CAOS run. We call such agents *static*, and we allow code in agent message handlers to reference such agents by name. Before an agent begins running, each symbolic reference is resolved by the CAOS runtimes.

There are a number of additional characteristics; most of these are used by CAOS internally, and we will document these in the next chapter.

The basic form for declaring a CAOS agent is `defagent`. It has the form illustrated by Figure 4.1. The first element in each sublist is a keyword; there are a number of defined keywords, and their use in an agent declaration is strictly optional. An agent inherits the union of the keyword values of its parents for any unspecified keyword. Of those keywords which *are* specified, some are combined with the union of the keyword values of the agent's parents, and others supersede the values in the parents. Figure 4.2 contains the declaration of the emitter agent, one of the most complex examples in ELINT.

As we discuss in the next chapter, `defagent` forms are translated by CAOS into Flavors `defflavor` forms [4]. CAOS messages are then defined using the `defmethod` function of ZETALISP. These methods are free to reference the local variables declared in the `defagent` expression.

4.2 Initialization of agents

The initial CAOS configuration is specified by the `caos-initialize` operator, which takes the form illustrated by figure 4.3; for example, figure 4.4 is ELINT's initialization form.

The first portion of the form creates the static agents. In figure 4.4, a static agent named `el-gotcha-handler-1`, an instance of the class `el-observation-handler`, is created on the CARE site at coordinates $(1,2)$ in the processor grid.

The second portion of the form is a list of LISP expressions to be evaluated sequentially when CAOS's initialization phase is complete. Each expression is intended to send a message to one of the static agents declared in the first part of the form. These messages serve to initialize the application; in figure 4.4, the initialization messages open log files and start the processing of ELINT observations.

Agents may also be created dynamically. The `create-agent-instance` function accepts an agent class name and a location specification;[1] the `remote-address` of the newly-created agent is returned. While dynamically created agents may *not* be referenced symbolically, their `remote-address`'s may be exchanged freely.

[1] Currently, agents may be created at or near specified CARE sites. CAOS makes no attempt at dynamic load balancing.

```
(defagent el-emitter (process-agenda-agent)
  (localvars
    (process-agenda '(el-undo-collection-id-error
                      el-change-cluster-association
                      el-emitter-update-on-time-tick
                      el-initialize-emitter
                      el-update-emitter-from-observation))
    (last-observed -1000000)
    (cluster-manager 'cluster-manager-0)
    manager
    id
    type
    observed
    fixes
    last-heading
    last-mode
    confidence
    cluster
    new-observations-since-time-tick-flag
    id-errors
    gc-flag)
  (messages
    el-update-emitter-from-observation
    el-initialize-emitter
    el-change-cluster-association
    el-undo-collection-id-error)
  (symbolically-referenced-agents
    el-collection-reporter-0
    el-correlation-reporter-0
    el-threat-reporter-0
    el-cluster-manager-0
    el-cluster-manager-1
    el-cluster-manager-2
    el-big-ear-handler
    el-gotcha-handler
    el-emitter-trace-reporter-0))
```

Figure 4.2: The emitter agent

```
(caos-initialize
  ((agent - name_1 agent - class site - address)
    ...)
  ((initial - message_1)
    ...))
```

Figure 4.3: The basic CAOS initialization form

4.3 Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee that messages reach their destinations: due to excessive message traffic or processing element failure, messages may be delayed or lost during routing. It is the responsibility of the application program to detect and recover from lost messages. Commensurate with the facilities provided by CARE, messages may be tagged with routing priorities; however, higher priority messages are not guaranteed to arrive before lower-priority messages sent concurrently.

Two classes of messages are defined: those which return values (called *value-desired* messages), and those which do not (called *side-effect* messages). The value-desired-messages are made to return their values to a special cell called a *future*. Processes attempting to access the value of a future are blocked until that future has had its value set. It is possible for the value of a future to be set more than once, and it is possible for there to be multiple processes awaiting a future's value to be set.[2]

4.3.1 Sending messages

The CARE primitive **post-packet**, which sends a packet from one process to another, is employed in CAOS to produce three basic kinds of message sending operations:

post: *The* **post** *operator sends a side-effect message to an agent. The sending process supplies* the name or pointer to the target agent, the message routing priority, the message name and arguments. The sender continues executing while the message is delivered to the target agent.

post-future: The **post-future** operator sends a value-desired message to the target agent. The sending process supplies the same parameters as for **post**, and is returned a pointer to the future which will eventually by set by the target agent. As for **post**, the sender continues executing while the message is being delivered and executed remotely.

A process may later check the state of the future with the **future-satisfied?** operator, or access the future's value with the **value-future** operator, which will block the process until the future has a value.

post-value: The **post-value** operator is similar to the **post-future** operator; however, the sending process is delayed until the target agent has returned a value. **post-value** is defined in terms of **post-future** and **value-future**.

[2] Futures were also used in QLisp and Multilisp [9]. The HEP Supercomputer [6] implemented a simple version of futures as a process synchronization mechanism.

```
(caos-initialize
  ((el-observation-reader-0 el-observation-reader (2 2))
   (el-big-ear-handler-1 el-observation-handler (1 1))
   (el-big-ear-handler-2 el-observation-handler (1 1))
   (el-gotcha-handler-1 el-observation-handler (1 2))
   (el-gotcha-handler-2 el-observation-handler (1 2))
   (el-emitter-manager-0 el-emitter-manager (2 1))
   (el-emitter-manager-1 el-emitter-manager (2 2))
   (el-collection-reporter-0 el-collection-reporter (1 2))
   (el-correlation-reporter-0 el-correlation-reporter (1 3))
   (el-threat-reporter-0 el-threat-reporter (1 3))
   (el-emitter-trace-reporter-0 el-emitter-trace-reporter
                                (3 2))
   (el-cluster-trace-reporter-0 el-cluster-trace-reporter
                                (3 1))
   (el-cluster-manager-0 el-cluster-manager (2 1)))
  ((post el-observation-reader-0 nil
        'el-open-observation-file
         *elint-data-file*)
   (post el-collection-reporter-0 nil
         'el-initialize-reporter t
         "elint:reports;collections.output")
   (post el-correlation-reporter-0 nil
         'el-initialize-reporter t
         "elint:reports;correlations.output")
   (post el-threat-reporter-0 nil
         'el-initialize-reporter t
         "elint:reports;threats.output")
   (post el-emitter-trace-reporter-0 nil
         'initialize-trace-reporter t
         "elint:reports;emitter.traces")
   (post el-cluster-trace-reporter-0 nil
         'initialize-trace-reporter t
         "elint:reports;cluster.traces")))
```

Figure 4.4: The initialization declaration for ELINT.

4.3.2 Detecting Lost Messages

It is possible to detect the loss of value-desired messages by attaching a timeout to the associated future. The functions `post-clocked-future` and `post-clocked-value` are similar to their untimed counterparts, but allow the caller to specify a *timeout* and *timeout action* to be performed if the future is not set within the timeout period. Typical actions include setting the future's value with a default value, or resending the original message using the `repost` operator.

4.3.3 Sending to Multiple Agents

There exist versions of the basic posting operators which allow the same message to be sent to multiple agents.[3] `multipost` sends a side effect message to a list of agents; `multipost-future` and `multipost-value` send a value-desired message to a list of agents. In the latter case, the associated future is actually a list of futures; the future is not considered set until all target agents have responded. The value of such a message is an association-list; each entry in the list is composed of an agent name or `remote-address` and the returned message value from that agent. There exist clocked versions of these functions (called, naturally, `multipost-clocked-future` and `multipost-clocked-value`) to aid in detecting lost multicast messages.

4.4 Communications Between Processes

Processes in each agent communicate using the shared local variables declared in the agent. Besides sharing previously computed results this way, processes may also share the results of ongoing computations.

Consider the following scenario: within an agent, some process is currently computing some answer. At the same time, another process begins executing, and realizes somehow that the answer it needs to compute is the same answer the other process is already computing. The second process could take one of two actions: it could continue computing the answer, even though this would mean redundant work, or it could wait for the first process to complete, and return its answer. The second approach is feasible, but it does tie up resources in the form of an idle process.

The CAOS operators `attach` and `my-handle` offer a third alternative solution. If a process knows it may ultimately produce an answer needed by more than one requesting agent, it obtains its "handle" (Section 5.4) by calling `my-handle`, and places it in a table for other processes to reference. Any other process wishing to return the same answer as the first calls `attach`, with the first process's handle as argument. The first process returns its answer to all requesting agents waiting for answers from the other processes, and the other processes return no value at all.

4.5 What CAOS Offers Over CARE

CAOS is a large system. It is reasonable to ask what advantages there are to programming in CAOS as opposed to programming in CARE. We believe there are three major advantages:

[3]Neither CAOS nor CARE currently support a *predicated multicast* mode, wherein messages would sent to all agents satisfying a particular predicate; messages can only be sent to a fully-specified list of agents.

Clarity: The framework in which an agent is declared makes explicit its storage requirements and functional behavior. In addition, the agent concept is a helpful abstraction at which to view activity in a multiprocessing software architecture. The concept lets us partition a flat collection of processes on a site into groups of processes attached to agents on a site. CAOS guarantees the only interaction between processes attached to different agents is by message-passing.

Convenience: The programmer is freed from interfacing to CARE's low-level communications primitives. As we said earlier, CAOS is basically an operating system, and as such, it shields the programmer from the same class of details a conventional operating system does in a conventional hardware environment.

Flexibility: Currently, CARE schedules processes in a strict first-in, first-out manner. CAOS, on the other hand, can implement arbitrary scheduling policies (though at a substantial performance cost; we discuss this in Chapter 6).

Chapter 5

The Runtime Structure of CAOS

CAOS is structured around three principal levels: site, agent, and process. Two of these levels–site and process–reflect the organization of CARE; the remaining (agent) level is an artifact of CAOS. We discuss first the general design principles underlying CAOS, and then describe in greater detail the functions and structure of each of CAOS's levels. Appendix B offers a complete guide to the algorithms and data structures employed in CAOS.

5.1 General Design Principles

The implementation of CAOS described in this paper is written in ZETALISP, a dialect of Lisp which runs on a number of commercially available single-user Lisp workstations. ZETALISP includes an object-oriented programming tool, called Flavors, which has proved to be a very powerful facility for structuring large Lisp applications.

In Flavors, the behavior of an object is described by templates known as *classes*. An *instance*, a representation of an individual object, is created by instantiating a class. Instances respond to messages defined by their class, and contain static local storage in the form of *instance variables*. Classes are defined within an inheritance network; each instance contains the instance variables and responds to the messages defined in its class, as well as those of the classes from which its class inherits.

An appropriate usage for Flavors is the modelling of the behavior of objects in some (not necessarily real) world. For example, CAOS site and agents structures are realized as Flavors instances. The characteristics to be modelled are codified in instance variables and message names. In a well-designed application, messages and variables are consistently named; thus, the implementation of a particular behavior is totally encapsulated in the anonymous function which responds to a message.

5.1.1 Extending the Notion

In some sense, a Flavors instance is an abstract data type. The instance holds state, and provides advertised, public interfaces (messages) to functions which change or access its state. The internal data representation and implementations of the access functions are private.

In Flavors, the abstract data type notion is unavailable within an individual instance. Frequently, the individual instance variables hold complex structures (such as dictionaries and priority queues) which ought to be treated as abstract data types, but there exist no common means within the standard Flavors mechanism for doing so.

CAOS, however, supports such a mechanism, by providing a means of sending messages to instance variables (rather than to the instances themselves). The instance variables are thus able to store anonymous structures, which are initialized, modified, and accessed through messages sent to the variable. Similar mechanisms exist in the Unit Package [14] and in the STROBE system [13], both frameworks for representing structured knowledge.

The CAOS environment includes a number of abstract data types which were found to be useful in supporting its own implementation. The most commonly used are:

Dictionary: The dictionary is an association list. It responds to `put`, `get`, `add`, `forget`, and `initialize` messages.

Sorted Dictionary: The sorted-dictionary is also implemented as an association list, and responds to the same messages as does the standard dictionary. However, the sorted-dictionary invokes a user-supplied priority function to merge new items into the dictionary (higher-priority items appear nearer the front of the dictionary). This dictionary is able to respond to the `greatest` message, which returns the entry with the highest priority, and to the `next` message, which returns the entry with the next-highest priority as compared to a given entry.

The sorted-dictionary is used primarily to hold time-indexed data which may be collected out-of-order (e.g. when data for time $n + 1$ may arrive before data for time n).

Hash Dictionary: The hash-dictionary is implemented with a hash table, and responds to the same messages as the unsorted association list dictionary.

Queue: The queue data type is a conventional first-in, first-out storage structure. The put message enqueues an item on the tail of the queue, while the `get` message dequeues an item from the head of the queue.

Priority Queue: The priority-queue data type supports a dynamic heapsort, and is implemented as a partially-ordered binary tree. It responds to `put`, `get`, and `initialize` messages. Associated with the queue is a function which computes and compares the priority of two arbitrary queue elements; this function drives the rebalancing of the binary tree when elements are added or deleted.

Monitor: A monitor provides mutual exclusion within a dynamically-scoped block of Lisp code. It is similar in implementation to the monitors of Interlisp-D and Mesa [10].

If the monitor is unlocked, the `obtain-lock` message stores the caller's process id as the monitor's owner, and marks the monitor as locked; otherwise, if the monitor is locked, the `obtain-lock` message places the caller's process id on the tail of the monitor's waiting queue, and suspends the calling process.

The `release-lock` message removes the process id from the head of the monitor's waiting queue, marks the monitor's owner to be that id, and reschedules the associated process.

Monitors are normally accessed using the `with-monitor` form, which accepts the name of an instance variable containing a monitor, and which cannot be entered until the calling process obtains ownership of the monitor. The `with-monitor` form guarantees ownership of the monitor will be relinquished when the calling process leaves the scope of the form, even if an error occurs.

5.2 The CAOS Site Manager

The site manager consists of a Flavors instance containing information global to the site–information needed by all agents located on the site. In addition, the site manager includes a CARE-level process which performs the functions of creating new agents and translating agent names into agent addresses, as described below.

The following instance variables are part of the site manager:

`incoming-stream`: This instance variable contains the CARE input stream address on which the site manager process listens for requests. Agents needing to send messages to their site manager may reference this instance variable in order to discover the address to which to direct site requests.

`static-agent-stream-table`: This instance variable is a dictionary which maps agent names into the CARE streams which may be used to communicate with the agents. The entries in this dictionary reflect statically-created agents; new entries are added as the result of `new-initial-agent-online` messages directed to the site (see below). The dictionary is used to resolve agent name-to-address requests from agents created locally.

`unresolved-agent-stream-table`: The site manager keeps track of agent names it is not able to translate to addresses by placing unsatisfiable `request-symbolic-reference` requests in this dictionary. The keys of the dictionary are unresolvable agent names. As the agent names become resolvable, the unsatisfied requests are satisfied, and the corresponding entries are removed from the dictionary.

After the initialization phase of a CAOS application has completed, there will be no entries in this dictionary in any of the sites.

`local-agents`: This instance variable is a dictionary whose keys are the names of agents located on the site, and whose values are pointers to the Flavors instances which represent each agent. `local-agents` is used only for debugging and status-reporting purposes.

`free-process-queue`: When a CARE process which was created to service a request finishes its work, it tries to perform another task for the agent in which it was created. If the agent has no work to do, the process suspends itself, after enqueuing identifying information in this instance variable, which holds a queue abstract data type. When any agent on the same site needs a new process to service some request, it checks this queue first; if there are any suspended (free) processes waiting in this queue, it dequeues one and gives it a task to perform. If this queue is empty, the agent asks CARE to create a new process.

The site manager responds to the following messages:

`new-initial-agent-online`: As each static agent starts running during initialization of a CAOS run, it broadcasts its name and CARE input stream to every site in the system, using this message. The correspondence between the sending agent's name and address is placed in the `static-agent-stream-table` dictionary for future reference by agents located on the receiving sites. If any agents have placed requests for this new agent in the `unresolved-agent-stream-table`, messages containing the new agent's name and address are sent to the waiting agents.

`request-symbolic-reference`: Whenever a static agent is created, it runs an initialization function, which among other tasks, caches needed agent name-to-address translations. For each translation, the agent sends this message to its site manager. If the site manager can resolve the name upon receipt of the message, it responds immediately; otherwise, it queues the request in the `unresolved-agent-stream-table`, and defers answering until it is able to satisfy the request. The requesting agents waits until it has received the answer before requesting another translation.

`make-new-agent`: This message is sent to a site to cause a new agent to be created during the course of a CAOS run. The site manager creates the new (dynamic) agent and returns the agent's input stream to the sender of this message. The newly-created agent is *not* placed in the `static-agent-stream-table`; thus, the only way to advertise the existence of such a dynamically-created agent is by the creator of an agent passing the returned input stream to other agents.

5.3 The CAOS Agent

As discussed above, CAOS agents are implemented as Flavors instances. Their class definitions are defined by translating `defagent` expressions into `defflavor` expressions. CAOS itself defines two basic agent classes: `vanilla-agent` and `process-agenda-agent`. `vanilla-agent` defines the minimal agent; `process-agenda-agent` is defined in terms of `vanilla-agent`, but adds the ability to assign priorities to messages.[1] These basic agents are fully-functional, but lack domain-specific "knowledge," and cannot be used directly in problem solving applications.

As stated in the previous chapter, a CAOS agent is a multiple-process entity. Most of these processes are in created in the course of problem-solving activity; we refer to these as *user processes*. At runtime, however, there are always two special processes associated with each CAOS agent. One of these processes monitors the CARE stream by which the agent is known to other agents. The other participates in the scheduling of user processes. We shall refer to the first of these processes as the agent *input monitor*, and to the second of these processes as the agent *scheduler*. We explain in detail the functioning of these two processes in the next section.

We describe here the role of important instance variables in a basic CAOS agent:

[1] This is important for applications in which one agent must respond rapidly to a posting from another agent. Assigning a message a high priority will cause that message to be processed ahead of any other messages with lower priorities.

self-address: This instance variable is an analogue of Flavors' **self** variable. Whereas **self** is bound to the Flavors instance under which a message is executing, **self-address** is bound to the stream of the agent under which a CAOS message is executing. Thus, an agent can post a message to itself by posting the message to **self-address**.

runnable-process-stream: This instance variable points to the stream on which the scheduler process listens. Processes which need to inform the scheduler of various conditions do so by sending CARE-level messages to this stream.

running-processes: This variable holds the list of user processes which are currently executing within the agent. The current CARE architecture supports only a single evaluator on each site. CAOS tries to keep a number of user processes ready to execute at all times; thus, the single CPU is kept as busy as possible.

runnable-process-list: A priority queue containing the runnable user processes. As a process is entered on the queue, its priority is calculated to determine its ranking in the partial ordering. There are two available priority evaluation functions: the first computes the priority based solely on the time the process entered the system; the second considers the assigned priority of the executing message before considering the entry time of the process. These two functions are used to implement the scheduling algorithms of the **vanilla-agent** and the **process-agenda-agent**, respectively.

scheduler-lock: The scheduler data structures are subject to modification by any number of processes concurrently. The **scheduler-lock** is a monitor which provides mutual exclusion against simultaneous access to the scheduler database.

5.4 The CAOS Process

In this section, we describe the mechanism by which CAOS user processes are scheduled for execution on CARE sites. User processes are created in response to messages from other agents. Associated with each user process is a data structure called a **runnable-item**. The **runnable-item** contains the following fields:

message-name, -args, -id, -answer-targets: These fields store the information necessary to handle a message request and send the resulting answer back to the proper agents.

for-effect: This field is a boolean, and indicates whether the message is being executed for effect or value. This corresponds directly to the source of the message coming from a **post** operation or a **post-future** operation.

state: This field indicates the state of the process. The possible states that a process may enter, and the finite state machine which defines the state transition are discussed in the next section.

context: This field contains a pointer to the CARE stream upon which the process waits when it not runnable. A process (such as the scheduler) wishing to wake another process simply sends a message to this stream. The suspended process will thus be awakened (by CARE).

`time-stamp`: This field contains the time at which the process entered the system. It is used by the functions which calculate the execution priority of processes.

The CAOS scheduler's only handle on a process is the process's `runnable-item`. In fact, the only communication between a user process and the CAOS scheduler consists of the exchange of `runnable-item`'s.

5.5 Flow of Control

In the following, we detail how a user process, the CAOS input monitor, and the CAOS scheduler interact to process a message request from a remote agent. For purposes of exposition, we assume the following sequence of events:

1. An agent, `agent-1`, executes a `post` operation, with `agent-2` as the target. The posting is for the message named `message-a`.
2. `agent-2` receives and executes the posting. In order to complete the execution of `message-a`, it must perform a `post-value` operation to a third agent, `agent-3`.

We begin at the point where `agent-1` has performed its `post` operation.

5.5.1 Input Processing

The input monitor process handles requests and responses from remote agents. When the message from `agent-1` enters `agent-2`, its input monitor creates a new `runnable-item` to hold the state of the request. The message name, arguments, id, and answer targets are copied from the incoming message into the `runnable-item`. The `runnable-item`'s state is set to `never-run`, and its time stamp is set to the current time. In order to queue the message for execution, the input monitor takes one of two actions.

If the agent's `runnable-process-list` is empty, the `runnable-item` is sent in a message to the agent scheduler process (by sending the item in a message to the stream whose address is found in the agent's `runnable-process-stream` instance variable). When the agent's `runnable-process-list` is empty, the scheduler process is guaranteed to be waiting for messages sent to the scheduler stream, and hence, will be awakened by the message sent from the input monitor. The scheduler then computes the priority of the message, and places the `runnable-item` in its `runnable-process-list`.

If the agent's `runnable-process-list` is *not* empty, the input monitor computes the message's priority and places the `runnable-item` on the `runnable-process-list` itself. When the queue is not empty, it is guaranteed that the scheduler will examine the queue sometime in the future to make scheduling decisions; thus, it is not necessary to send any messages to the scheduler to inform it of the existence of new processes.

5.5.2 Creating Processes

Eventually, the newly-created `runnable-item` will reach the head of `agent-2`'s `runnable-process-list`. At this time, there is still no process associated with the item, so the scheduler creates a process using the facilities of CARE, adds the process to the `running-processes` list, and passes it its `runnable-item`. The process will eventually gain control of the evaluator, and will set the state of its `runnable-item` to `running`. It then begins executing the requested posting.

5.5.3 Requesting Remote Values

At some point, the process executing on `agent-2` requires a value from `agent-3`, and performs a `post-value` operation to acquire it. The process looks up the address of `agent-3`, and posts a message which contains the appropriate message name, arguments, id, and answer target. The `message-id` unambiguously identifies the `future` upon which the process will be waiting for the value to be returned. The answer target is the agent's own `self-address`; when the answer is received by the input monitor process, it will be forwarded to the appropriate future, and the process will be reawakened.

In the meantime, the process sets its state to **suspended**, removes its `runnable-item` from the `running-processes` list, and appends it to the list of processes already waiting for the future to be satisfied. If the `runnable-process-list` is not empty, the suspending process wakes the process at the head of the queue.[2] The suspending process then waits for a message on its wakeup stream, the stream whose address is in the `context` field of its `runnable-item`.

5.5.4 Answer Processing

Some time later, `agent-3` will have completed its computations, and will have returned the desired answer to `agent-2`. The answer will be received by `agent-2`'s input monitor process, which will recognize the input as a value to be placed in a future. The input monitor sets the value field of the appropriate future, and moves the `runnable-items` of the processes waiting on the future to the `runnable-process-list`.

If the queue was previously empty, the agent must have been (or will soon be) entirely idle; thus, the `runnable-items` are sent to the scheduler in a message, causing the scheduler to be reawakened. If the queue was not previously empty, the agent must be busy, so the items are simply added to the queue according to their priorities. In both cases, the `runnable-items` are placed in the `runnable` state.

5.5.5 Reawakening Suspended Processes

When the `runnable` `runnable-item` reaches the head of `agent-2`'s `runnable-process-list`, a message (which contains no useful information) is sent to its associated process's wakeup stream. As a result, process eventually wakes up, gains control of the evaluator, and sets its state to `running`.

[2] In effect, the process takes on the role of the scheduler. Although the system would continue to work with only a designated scheduler process performing scheduler duties, this arrangement permits scheduling to take place with minimal latency. As a result, fewer evaluator cycles are wasted waiting for the scheduler process to run the next user process.

5.5.6 Completing Computation

A process may perform any number of `post`, `post-future`, or `post-value` operations during its lifetime. Eventually, however, the process will complete, having computed a value which may or may not be sent back to the requesting agent. If the process was suspended for any portion of its lifetime, another process may have attached to it; in this case, the process may have more than one requesting agent to which to return an answer.

Before the process terminates, it examines the head of the `runnable-process-list`. If the queue is empty, the process simply goes away. If the `runnable-item` at the head of the queue is `runnable`, it sends the appropriate message to awaken the associated process. Finally, if the item is `never-run`, the process makes itself the process associated with this new `runnable-item`, and executes the new message in its own context.[3] Barring this possibility, the process "queues" itself on a free process queue associated with the site manager; when a new process is needed by an agent on the site, one is preferentially removed from this queue and recycled before a entirely new process is created. This way, processes, which are expensive to create, are reused as often as possible.

[3]This is another situation in which an application process performs scheduling duties.

Chapter 6

Results and Conclusions

The CAOS system we have described has been fully implemented and is in use by two groups within the Advanced Architectures Project. CAOS runs on the Symbolics *3600* family of machines, as well as on the Texas Instruments *Explorer* Lisp machine. ELINT, as described in Section 3.2, has also been fully implemented. We are currently analyzing its performance on various size processor grids and at various data rates.

6.1 Evaluating CAOS

CAOS is a rather special-purpose environment, and should be evaluated with respect to the programming of concurrent real-time signal interpretation systems. In this chapter, we explore CAOS's suitability along the following dimensions:

- Expressiveness
- Efficiency
- Scalability

6.1.1 Expressiveness

When we ask that a language be suitably *expressive*, we ask that its primitives be a good match to the concepts the programmer is trying to encode. The programmer shouldn't need to resort to low-level "hackery" to implement operations which ought to be part of the language. We believe we have succeeding in meeting this goal for CAOS (although to date, only CAOS's designers have written CAOS applications). Programming in CAOS is programming in Lisp, but with added features for declaring, initializing, and controlling concurrent, real-time signal interpretation applications.

6.1.2 Efficiency

CAOS has a very complicated architecture. The lifetime of a message, as described in Section 5.5, involves numerous processing states and scheduler interventions. Much of this complexity derives from the desire to support alternate scheduling policies within an agent. The cost of this complexity is approximately one order of magnitude in processing latency. For the common settings of simulation parameters, CARE messages are exchanged in about 2–3 milliseconds, while CAOS messages require about 30 milliseconds. It is this cost which forces us to decompose applications coarsely, since more fine-grained decompositions would inevitably require more message traffic.

We conclude that CAOS does not make efficient use of the underlying CARE architecture. A compromise, which we are just beginning to explore, would be to avoid the complex flow of control described in Section 5.5 in agents whose scheduling policies are the same as CARE's (FIFO). In such agents, we could reduce the CAOS runtimes to simple functional interfaces to CARE. We anticipate such an approach would be much more efficient.

6.1.3 Scalability

A system which scales well is one whose performance increases commensurately with its size. Scalability is a common metric by which multiprocessor hardware architectures are judged: does a 100-processor realization of a particular architecture perform 10 times better than a 10-processor realization of the same architecture? Does it perform 5 times better? Only just as well? Or *Worse?* In hardware systems, scalability is typically limited by various forms of *contention* in memories, busses, etc. The 100-processor system might be slower than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support 10 processors.

We ask the same question of a CAOS application: does the throughput of ELINT, for example, increase as we make more processors available to it? This question is critical for CAOS-based real-time interpretation systems; our only means of coping with arbitrarily large data rates is by increasing the number of processors. Section 6.2 discusses this issue in detail.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to its scaling are *(1)*, increased software contention, such as inter-pipeline bottlenecks described in Section 3.1.2, and *(2)*, increased hardware contention, such as overloaded processors and/or communication channels. Software contention can be minimized by the design of the application. Communications contention can be minimized by executing CAOS on top of an appropriate hardware architecture (such as that afforded by CARE); CAOS applications tend to be coarsely decomposed–they are bounded by computation, rather than communication–and thus, communications loading has never been a problem.

Unfortunately, processor loading remains an issue. A configuration with poor *load balancing*, in which some processors are busy, while others are idle, does not scale well. Increased throughput is limited by contention for processing resources on overloaded sites, while resources on unloaded sites go unused. The problem of automatic load balancing is not addressed by CAOS; agents are assigned to processing sites on a round-robin basis, with no attempt to keep potentially busy agents apart.

ELINT *Performance Dimension*	*Control Type/Grid Size*					
	NC	CC	CC	CT	CT	CT
	4 x 4	4 x 4	6 x 6	2 x 2	4 x 4	6 x 6
FALSE ALARMS	1	0	0	0	0	0
REINCARNATION	49	42	2	0	0	0
CONFIDENCE LEVEL	19	20	90	89	93	95
FIXES	48	42	99	100	100	100
FUSION	0	0	77	85	88	89

Table 6.1: Quality of ELINT performance of various grid sizes and control strategies (1 ELINT time unit = 0.1 seconds).

6.2 Evaluating ELINT Under CAOS

Our experience with ELINT indicates the primary determiner of throughput and answer-quality is the strategy used in making individual agents cooperate in producing the desired interpretation. Of secondary importance is the degree to which processing load is evenly balanced over the processor grid. We now discuss the impact of these factors on ELINT's performance.

The following three strategies were used in our experiments:

NC: This strategy represents limited inter-agent control. No attempt is made to prevent concurrent creation of multiple copies of the "same" agent (this possibility arises when multiple requests to create the agent arrive simultaneously at a single manager). As a result, multiple, non-communicating copies of an abstraction pipeline are created; each receives a only portion of the input data it requires. The NC strategy was expected to produce poor results, and was intended only as a baseline against which to compare more realistic control strategies.

CC: In this strategy, the manager agents assure that only one copy of a agent is created, irrespective of the number of simultaneous creation requests; all requestors are returned pointers to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce correct high-level interpretations.

CT: The CT ("creation *and* time control") strategy was designed to manage skewed views of real-world time which develop in agent pipelines. In particular, this strategy prevents an `emitter` agent from deleting itself when it has not received a new observation in a while, yet some `observation-handler` agent has sent the `emitter` an observation which it has yet to receive.

Table 6.1 illustrates the effects of various control strategies and grid sizes. The table presents six performance attributes by which the quality of an ELINT run is measured.

False Alarms: This attribute is the percentage of `emitter` agents that ELINT should not have hypothesized as existing.

ELINT was not severely impacted by false alarms in any of the configurations in which it was run.

Control Type	*Simulated Time (sec)*		
	2 × 2	4 × 4	6 × 6
NC		> 11.19[a]	
CC		10.87	5.12
CT	11.80	8.10	4.17

[a]This run was far from completion when it was halted due to excessive accumulated wall-clock time.

Table 6.2: Simulated time required to complete an ELINT run (1 ELINT time unit = 0.1 seconds).

Control Type	*Message Count*		
	2 × 2	4 × 4	6 × 6
NC		> 16118	
CC		7375	
CT	4516	4703	4616

Table 6.3: Number of messages exchanged during an ELINT run (1 ELINT time unit = 0.1 seconds).

GRID SIZE	1 × 1	2 × 2	3 × 3	4 × 4	5 × 5	6 × 6
SIMULATED TIME (sec)	9.42	3.20	1.49	0.74	0.52	0.56

Table 6.4: Overall Simulation Times for CT Control Strategy (1 ELINT time unit = 0.01 seconds, debugging agents turned off).

Reincarnation: This attribute is the percentage of recreated `emitter` agents (*e.g.*, `emitters` which had previously existed but had deleted themselves due to lack of observations). Large numbers of reincarnated `emitters` indicate some portion ELINT is unable to keep up with the data rate (*i.e.*, the data rate may be too high globally, so that all `emitters` are overloaded, or the data rate may be too high locally, due to poor load balancing, so that some subset of the `emitters` are overloaded).

The CT control strategy was designed to prevent reincarnations; hence, none occurred when CT was employed on any size grid. When CC was used, only the 6×6 grid was large enough for ELINT to keep up with the input data rate.

Confidence Level: This attribute is the percentage of correctly-deduced confidence levels of the existence of an `emitter`.

The correct calculation of confidence levels depends heavily on the system being able to cope with the incoming data rate. One way to improve confidence levels was to use a large processor grid. The other was to employ the CT control strategy, since fewer reincarnations result in fewer incorrect (*e.g.*, too low) confidence levels.

Fixes: This attribute is the percentage of correctly-calculated fixes of an `emitter`.

Fixes can be computed when an `emitter` has seen at least two observations in the same time interval. If an `emitter` is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation maximized the correct calculation of fix information.

Fusion: This attribute is the percentage of correct clustering of `emitter` agents to `cluster` agents.

The correct computation of fusion appeared to be related, in part, to the correct computation of confidence levels. The fusion process is also the most knowledge-intensive computation in ELINT, and our imperfect results indicate the extent to which ELINT's knowledge is incomplete.

We interpret from Table 6.1 that control strategy has the greatest impact on the quality of results. The CT strategy produced high-quality results irrespective of the number of processors used. The CC strategy, which is much more sensitive to processing delays, performed nearly as well only on the 6×6 processor grid. We believe the added complexity of the CT strategy, while never detrimental, is only beneficial when the interpretation system would otherwise be overloaded by high data rates or poor load balancing.

Tables 6.2 and 6.3 indicate that cost of the added control in the CT strategy is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulation time is reduced (In Table 6.2, the last observation is fed into the system at 3.6 seconds; hence, this is the minimum possible simulated run time for the interpretation problem).

Finally, Table 6.4 illustrates the effect of processor grid size when the CT control strategy is employed. This table was produced with the data rate set ten times higher than that used to produce tables 6.1–6.3; the minimum possible simulated run time for the interpretation problem is 0.36 seconds. The speedup achieved by increasing the processor grid size is nearly linear with the square root of the size; however, the 6×6 grid was slightly slower than the 5×5 grid. In this last case, we believe the data rate was not high enough to warrant the additional processors.

6.3 Unanswered Questions

CAOS has been a suitable framework in which to construct concurrent signal interpretation systems, and we expect many of its concepts to be useful in our future computing architectures. Of principal concern to us now is increasing the efficiency with which the underlying CARE architecture is used. In addition, our experience suggests a number of questions to be explored in future research:

- What is the appropriate level of granularity at which to decompose problems for CARE-like architectures?
- What is the most efficient means to synchronize the actions of concurrent problem solvers when necessary?
- How can flexible scheduling policies be implemented without significant loss of efficiency? What is the impact on problem solving if alternate scheduling policies are not provided?

We have started to investigate these questions in the context of a new CARE environment. The primary difference between the original environment and the new environment is that the *process* is no longer the basic unit of computation. While the new CARE system still supports the use of processes, it emphasizes the use of *contexts*: computations with less state than those of processes.

When a context is forced to suspend to await a value from a stream, it is aborted, and restarted from scratch later when a value is available. This behavior encourages fine-grained decomposition of problems, written in a functional style (individual methods are small, and consist of a binding phase, followed by an evaluation phase).

In addition, CARE now supports arbitrary prioritization of messages delivered to streams. As a result, it is no longer necessary to include in CAOS its complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs between two and three orders of magnitude faster than the configuration described in this paper.

Acknowledgements

My thanks to Harold Brown, Bruce Delagi, and Reid Smith for reading and commenting on earlier drafts of this paper. Bruce Delagi, Sayuri Nishimura, Russell Nakano, and James Rice created and maintain the CARE environment. Harold Browr defined the behavior of the CAOS operators, ported ELINT from AGE to CAOS, and collected the results which appear in Section 6.2. Finally, I wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Lab for their excellent support of our computing environment.

Appendix A

Mergesort: A Simple CAOS Application

Mergesort is an efficient sorting algorithm. It is simple, and well-suited to a concurrent, message-passing implementation. As *mergesort* is not a real-time application, we need not be concerned with the effects of any data rate. Further, its run time is determined entirely by the size of the input; it is not sensitive to initial sorting of the data.

Our algorithm recursively subdivides the input list into two half-size lists, until lists of length 2 are obtained. These lists are then trivially sorted, and recombined in sorted order as the recursion is unwound. We exploit the concurrent CAOS architecture by implementing the recursion as **post-value** messages sent to other agents. Each processor contains a single **mergesort** agent. Agents are assigned in a globally *round-robin* order, and are created when necessary by a **mergesort-manager**; we employ one manager per column in the processor grid (this makes use of a natural invariant which lets us replicate managers—see our discussion of this approach within ELINT, in Section 3.2). The algorithm adapts automatically to different processor grids.

Table A.1 illustrates **mergesort**'s runtime on different processor grids and on various input lengths. **mergesort** is well-known to require $O(n \log n)$ time on a uniprocessor; similar analysis indicates **mergesort** should require $O(n)$ time on an "infinite" number of processors.[1] On a grid of size 1, **mergesort** implements a very expensive approach to a conventional *mergesort* (examine the leftmost column of the table); however, on a sufficiently large grid, the algorithm distributes computation across enough processors efficiently enough to achieve nearly $O(n)$ time (as seen in diagonal boundary of the table).

Table A.1 also illustrates the effects of choosing too small a grain-size for CAOS. **mergesort** is dominated by both communication and agent creation costs. It took substantially longer to sort an 8-element list on 4 processors than on 1 processor. Most of this time was spent waiting for answers from **mergesort-manager** agents.

[1] An infinite number of processors is a sufficient number to prevent any runnable "process" from having to wait for a free processor; in our implementation of *mergesort*, this number is $n/2$. Shapiro's implementation in Concurrent Prolog achieved $O(n)$ time with $O(\log n)$ processors [12].

n	Processor Grid Size					
	1	4	9	16	25	36
64	1414	912	756	640	537	514
32	803	606	466	432	471	
16	460	388	349	344		
8	274	397	242			
4	121	141				
2	31					

Table A.1: **mergesort** runtimes (in milliseconds) on various processor grids and input sizes.

A.1 The mergesort Source Code

This section contains the source code for **mergesort**. It is intended to show the flavor of programming in CAOS with a relatively simple example. We show first the code which declares and executes within the **mergesort** and **mergesort-manager** agents.

```
;;; Global variables controlling assignment of agents to sites
;;;
;;; If we were strict, this wouldn't be possible, since we're
;;; making use of the fact that memor in each site really isn't
;;; distributed.  However, we do this to force round-robin
;;; allocation.
(defconst *last-x* 1)
(defconst *last-y* 1)
(defconst *array-width* 1)
(defconst *array-height* 1)

;;; Define the basic mergesort agent
(defagent mergesorter (vanilla-agent)
  (documentation "An agent which can perform a level of mergesorting")
  (symbolically-referenced-agents
   ((mergesorter-1-1) mergesorter)
   ((mergesort-manager-1) mergesort-manager)
   ((mergesort-manager-2) mergesort-manager)
   ((mergesort-manager-3) mergesort-manager)
   ((mergesort-manager-4) mergesort-manager)
   ((mergesort-manager-5) mergesort-manager)
   ((mergesort-manager-6) mergesort-manager))
  (instance-vars
   (known-sorters vp-slot value nil datatype #$dictionary)
   (managers vp-slot value '((1 . mergesort-manager-1)
                             (2 . mergesort-manager-2)
                             (3 . mergesort-manager-3)
                             (4 . mergesort-manager-4)
                             (5 . mergesort-manager-5)
                             (6 . mergesort-manager-6))
             datatype #$dictionary))
  (messages-methods (mergesort :mergesort)))
```

```
;;; The initialize method clears the dictionary of site-agent
;;; mappings prior to the start of each run.
(defmethod (mergesorter :initialize) (&rest ignore)
  (send self 'known-sorters :initialize))

;;; The next-neighbor method returns a stream to a sorting agent
;;; which will perform half of the next lower-level recursive sort.
(defmethod (mergesorter :next-neighbor) ()
  (let ((next-location-site
         (multiple-value-bind (x y) (next-x-and-y)
           ;; x and y hold site coordinates for the next agent.
           (send (lookup-site x y) :care-site))))
    (let ((maybe-known-agent
           ;; check the dictionary for a site-agent mapping.
           (send self 'known-sorters :get next-location-site)))
      (cond (maybe-known-agent maybe-known-agent)
            (t (let ((next-location
                      (send next-location-site :location)))
                 ;; Don't know the mapping.  Ask a manager.
                 (send self 'known-sorters :put
                       next-location-site
                       (post-value (send self 'managers :get
                                         (first next-location))
                                   nil
                                   :new-agent (first next-location)
                                   (second next-location))))))))
```

```
(defmethod (mergesorter :mergesort) (&rest list)
  (cond ((eq (length list) 2)
         ;; Trivial case.  Lists of length 2.
         '(,(min (first list) (second list))
           ,(max (first list) (second list))))
        (t (let* ((first-neighbor (send self :next-neighbor))
                  (second-neighbor (send self :next-neighbor)))
             ;; Recurse: divide the list and sort both halves.
             ;; Use post-future to start each half.
             (first-future
              (lexpr-funcall #'post-future first-neighbor nil
                             :mergesort
                             (copylist (first-half list))))
             (second-future
              (lexpr-funcall #'post-future second-neighbor nil
                             :mergesort
                             (copylist (second-half list)))))
           ;; Combine the sorted sublists.
           ;; value-future blocks until the half finishes.
           (do ((e1 (value-future first-future)
                    (cond ((null e2) (cdr e1))
                          ((or (null e1) (> (first e1) (first e2)))
                           e1)
                          (t (cdr e1))))
                (e2 (value-future second-future)
                    (cond ((null e1) (cdr e2))
                          ((or (null e2) (> (first e2) (first e1)))
                           e2)
                          (t (cdr e2))))
                (result nil))
               ((and (null e1) (null e2)) result)
             (cond ((and e1 e2)
                    (setq result (nconc result
                                        (list (min (first e1)
                                                   (first e2))))))
                   (e1 (setq result (nconc result
                                           (list (first e1)))))
                   (t (setq result (nconc result
                                          (list (first e2))))))))))


;;; Function to maintain globally round-robin agent-site
;;; allocation.
(defun next-x-and-y ()
  (multiple-value-prog1 (values *last-x* *last-y*)
         (when (> (incf *last-x*) *array-width*)
           (setq *last-x* 1)
           (when (> (incf *last-y*) *array-height*)
             (setq *last-y* 1)))))

;;; Return the first half of a list.
(defun first-half (list)
  (loop for i from 1 to (// (length list) 2) as e in list
        collect e))

;;; Return the second half of a list.
(defun second-half (list) (nthcdr (// (length list) 2) list))

;;; Define the mergesort-manager.  These agents, located one
;;; per column in the processor grid, are responsible for
;;; creating new mergesort agents upon request.
(defagent mergesort-manager (vanilla-agent)
  (documentation "An agent to create other mergesorters")
  (instance-vars agent-array)
  (messages-methods (new-agent :new-agent)))

;;; The initialize method clears the manager's mapping of
;;; (x,y) coordinates to mergesort agent.
(defmethod (mergesort-manager :initialize) (max-x max-y)
  (setq agent-array (make-array (list (1+ max-x) (1+ max-y)))))

;;; The new-agent method returns the agent already at
;;; (x,y), or creates a new agent at (x,y) and returns it.
(defmethod (mergesort-manager :new-agent) (x y)
  (cond ((aref agent-array x y))
        (t (let ((the-new-agent (create-agent-instance
                                  'mergesorter
                                  (list x y))))
             (aset the-new-agent agent-array x y)
             the-new-agent))))
```

This next section of code is the CAOS initialization file which produced the runtime numbers displayed in Table A.1:

```
(defconst *the-original-list*
  '( 6  7  4  1  2  8  5  3 16 12  9 11 15 13 10 14
    32 22 30 21 28 19 26 18 24 31 22 29 20 29 25 17
    64 63 62 61 60 59 34 33 58 57 56 55 54 53 52 51
    50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35))

(defconst *the-current-list* nil)

(caos-initialize
 ((mergesorter-1-1 mergesorter (1 1))
  (mergesort-manager-1 mergesort-manager (1 1))
  (mergesort-manager-2 mergesort-manager (2 1))
  (mergesort-manager-3 mergesort-manager (3 1))
  (mergesort-manager-4 mergesort-manager (4 1))
  (mergesort-manager-5 mergesort-manager (5 1))
  (mergesort-manager-6 mergesort-manager (6 1)))
 ((with-open-file (log "x7:schoen.qsort;qsort.log" :write)
   (setq *the-current-list* *the-original-list*)
   (loop with start-time for j from 6 downto 1 do
         (format log "~&Sorting the list:~&~S"
                 *the-current-list*)
         (loop for i from 1 to j do
               (multipost-value
                 '(mergesort-manager-1 mergesort-manager-2
                   mergesort-manager-3 mergesort-manager-4
                   mergesort-manager-5 mergesort-manager-6)
                 nil :initialize i i)
               (post-value mergesorter-1-1 nil :initialize)
               (format log "~&Starting ~D processor sort at ~D"
                       (* i i) (caos-time))
               (setq start-time (caos-time))
               (lexpr-funcall #'post-value mergesorter-1-1 nil
                              :mergesort *the-current-list*)
               (format log "~&Finished at ~D.  That took ~D ms"
                       (caos-time)
                       (* (- (caos-time) start-time) 1.0e-5)))
         (setq *the-current-list* (first-half *the-current-list*))))))
```

We conclude with the log file produced by this mergesort execution:

```
Sorting the list:
(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14 32 22 30 21 28 19 26 18 24 31
22 29 20 29 25 17 64 63 62 61 60 59 34 33 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40 39 38 37 36 35)
Starting 1 processor sort at 9803527
Finished 1 processor sort at 151163188.  That took 1413.5966 ms
Starting 4 processor sort at 157430828
Finished 4 processor sort at 248600531.  That took 911.697 ms
Starting 9 processor sort at 254848384
Finished 9 processor sort at 330631571.  That took 757.83185 ms
Starting 16 processor sort at 337017977
Finished 16 processor sort at 401035492.  That took 640.1752 ms
Starting 25 processor sort at 407972369
Finished 25 processor sort at 461663705.  That took 536.9133 ms
Starting 36 processor sort at 468137724
Finished 36 processor sort at 519548649.  That took 514.10925 ms
Sorting the list:
(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14 32 22 30 21 28 19 26 18 24 31
22 29 20 29 25 17)
Starting 1 processor sort at 526138721
Finished 1 processor sort at 606424159.  That took 802.8544 ms
Starting 4 processor sort at 613038165
Finished 4 processor sort at 673645208.  That took 606.07043 ms
Starting 9 processor sort at 680223869
Finished 9 processor sort at 726796432.  That took 465.72562 ms
Starting 16 processor sort at 733697221
Finished 16 processor sort at 776848166.  That took 431.50943 ms
Starting 25 processor sort at 783605583
Finished 25 processor sort at 830669664.  That took 470.64078 ms
Sorting the list:
(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14)
Starting 1 processor sort at 837629049
Finished 1 processor sort at 883646903.  That took 460.17856 ms
Starting 4 processor sort at 890496880
Finished 4 processor sort at 929338867.  That took 388.41986 ms
Starting 9 processor sort at 936242285
Finished 9 processor sort at 971092553.  That took 348.5027 ms
Starting 16 processor sort at 978109126
Finished 16 processor sort at 1012524715.  That took 344.15588 ms
Sorting the list:
(6 7 4 1 2 8 5 3)
Starting 1 processor sort at 1019622193
```

```
Finished 1 processor sort at 1046974695.  That took 273.52502 ms
Starting 4 processor sort at 1054797480
Finished 4 processor sort at 1094519241.  That took 397.2176 ms
Starting 9 processor sort at 1101582612
Finished 9 processor sort at 1125786372.  That took 242.0376 ms
Sorting the list:
(6 7 4 1)
Starting 1 processor sort at 1132929674
Finished 1 processor sort at 1145004341.  That took 120.746666 ms
Starting 4 processor sort at 1152132853
Finished 4 processor sort at 1166264559.  That took 141.31706 ms
Sorting the list:
(6 7)
Starting 1 processor sort at 1173565420
Finished 1 processor sort at 1176647734.  That took 30.82314 ms
```

Appendix B

Implementing the CAOS Framework

This appendix is a guide to the source files which implement the CAOS system. The descriptions which follow are at a much greater level of detail than those in Chapter 5, and are intended primarily for readers of the source code, as a supplement to the embedded documentation. It is assumed that readers of this appendix have a familiarity with Lisp (principally ZETALISP or CommonLisp), and have read Chapter 5.

B.1 General Programming Issues

All data structures are implemented with the `defstruct` mechanism. `defstruct` accepts a description of the desired data structure, and produces a number of macro definitions which serve to create new instances of the structure, and access and modify fields of the structure. For example, a `ship` data structure may be defined as having fields `name`, `position`, and `course`. New instances of `ship`'s are created by calling `make-ship`; the fields of the ship structure are accessed by calling `ship-name`, `ship-position`, and `ship-course`. A field may be modified by embedding a field access function in a `setf` expression.

The CAOS system is intended for use in ZETALISP-compatible environments. The system was developed originally on the Symbolics *3600* family of workstations, and was later ported to the Texas Instruments *Explorer* workstation. These machines each support a ZETALISP programming environment, but are not completely source-code compatible.

Source-level incompatibilities are handled by use of the `#+` and `#-` reader macros. An occurrence of `#+Symbolics` in a source file causes the next s-expression to be read only when the file is being loaded into a Symbolics workstation; an occurrence of `#-Symbolics` prevents the following s-expression from being loaded into a Symbolics workstation. Similar read-time conditionals for the TI environment are introduced by `#+TI` and `#-TI` constructs.

B.2 Interface to CARE

In order to function properly under the CARE simulator, all CAOS code and CAOS applications must be loaded into the **care-user** symbol package. This package is defined to inherit from CARE those symbols (*e.g.*, functions, variables, and macros) which comprise the exported CARE programming interface.

B.2.1 CARE Data Structures

The following CARE-defined data structures are used CAOS:

remote-address [*Structure*]

A **remote-address** is the global encapsulation for the address of a data structure located on a particular processor. It may be thought of as extending the address space of a site with additional address bits that identify the site in the processor grid.

remote-address's contain two fields: **site** and **local**. The **site** field identifies the site on which the structure pointed to by the **local** field resides.

site [*Structure*]

A **site** represents one of the processing nodes in the grid. An instance of a **site** structure is actually an instance of a **site** flavor, and hence, fields of a **site** are accessed by sending Flavors messages. The following are messages relevant to CAOS: **:location**, which returns the (x, y) coordinate of the site in the grid; **:x-site**, which returns the x coordinate of the site; and **:y-site**, which returns the y coordinate.

queue [*Structure*]

A **queue** implements FIFO storage, and is used in a number of places within CARE. In particular, packets arriving on a CARE **stream** are stored in a **queue**. The **queue** structure has the following relevant fields: **length**, **body**, **tail**. The **length** field stores the number of entries which are currently in the queue; the **body** field points to a list which implements storage for the queue; the **tail** field points to the last element of the body of the queue, and allows new entries to be appended to the end of queue in $O(1)$ time (Access to the head of the queue also requires $O(1)$ time).

stream [*Structure*]

A **stream** is a virtual circuit which carries data (in the form of *packets*) between processes. Operations on streams are performed by the functions **post-packet** and **accept-packet**, which are described below. The **packets** field of a stream contains the queue of packets which have arrived on the stream. The **properties** field of a stream contains an arbitrary property list; CAOS uses the property list to store information to help the function which prints out streams in a human-readable fashion. Other fields of the stream are not relevant to CAOS.

process *[Structure]*

A **process** is the basic unit of computation in CARE. The innards of a **process** are of no concern to CAOS; however, it should be noted that the special variable *****care-process***** is always bound to the **process** structure of the process currently executing.

B.2.2 CARE Functions and Macros

The following functions and macros are used by CAOS:

post-packet &optional *form* **&key** ... *[Macro]*

The macro **post-packet** is used to create new streams and new processes, and to exchange messages between processes. If called with no arguments, it returns a new **stream** instance. All other **post-packet** options are controlled by the existence of various keywords in its argument list. When keyword arguments are supplied, the first argument to **post-packet** is evaluated to form the message to be sent.

The following keyword options are employed by CAOS:

to: The value of the **to** keyword is a stream or list of streams to which the message will be sent.

for: The value of the **for** keyword is a stream or list of streams. When the message is received remotely, the value of this keyword will appear in the **clients** field of the message.

for-new-stream, process: These two keywords always appear together in an argument list, and take no arguments. They are included in a call to **post-packet** to create new processes. The first argument in such a call is a form to evaluate remotely to start the process. This call also requires a **to** keyword argument, which must be a **remote-address**; the process is created on the site indicated by the **site** field.

The value of the call is a **stream**. A call to **accept-packet** on this stream will return a packet whose **value** field is the default stream supplied to the newly-created process.

after: The value of the **after** keyword is a time interval, in microseconds. When this keyword is supplied, the message will be delivered after a corresponding delay. The purpose of the keyword is to provide for a means of implementing *timeouts*. A process can cause a packet to be posted to a stream only after a specified interval; when this packet arrives, any processes waiting on the stream will be awakened. CAOS implements "clocked futures" using this mechanism.

tagged: The **tagged** keyword provides a means of tagging the message with a user-supplied value; its principal use is in debugging and message tracing.

with-packet-bindings *stream-form bindings* **&body** *forms* *[Macro]*

The `with-packet-bindings` macro evaluates *stream-form*, which must return a `stream`. It then picks the first packet from the stream (or blocks the calling process until a packet arrives), and (lambda) binds portions of the packet to the variables specified in *bindings*. The format of *bindings* is a list. The first variable name in the list is bound to the contents of the message; the second is bound to the clients of the message (*e.g.*, the streams specified by the `for` keyword in the call to `post-packet`). Additional variables may be bound to fields which are not relevant in the discussion of CAOS.

`accept-packet` *stream* [*Function*]

The macro `with-packet-bindings` is defined in terms of this function. `accept-packet` is called with *stream* bound to a `stream`, and returns the first packet waiting in the stream (or blocks the calling process until a packet is available).

`defprocess` [*Macro*]

The `defprocess` macro is syntactic sugar for `defun`. Any function which is to be the top-level of a CARE-process should be defined using `defprocess`. The last argument in the argument list of a function defined by `defprocess` will be bound to the default stream for the process; thus, any function defined with `defprocess` must have at least one argument.

B.3 The CAOS Support Environment

In Chapter 5, we described an extension to Flavors which implements abstract data type support for instance variables. The files `herbs.lisp`, `sage.lisp`, `datatype.lisp`, and `priority-queue.lisp` comprise the framework which includes abstract data type support. In addition, these files contain code which implements a sort of inheritance of default values of instance variables, and code which implements substructure for instance variables.

B.3.1 Herbs.Lisp

This file implements a form of inheritance of list-structured default values of instance variables. The Flavors class hierarchy forms a taxonomy; classes defined far from the root of the taxonomy are more specialized than those defined near the root. Within a class, methods can be combined with methods of the same name in ancestral classes in quite a few ways. Unfortunately, Flavors provides no means of combining inherited values.

Consider the example of Figure B.1. The Flavor class `flavor-3` is defined as a subclass of classes `flavor-1` and `flavor-2`. Both `flavor-1` and `flavor-2` define an instance variable called `iv-a`. What value does `flavor-3` inherit as the default for `iv-a`?

In normal Flavors, `flavor-3` would inherit `'(a b c)` as the default value. However, there are situations in which the proper value to inherit for `iv-a` might be `'(a b c d e f)`. The `defherb` macro, defined in `herbs.lisp`, enables this sort of inheritance.

Figure B.2 illustrates three possible inheritance modes for the default value of `iv-a` in `flavor-3`. In the first example, the default value of `iv-a` will be `'(a b c d e f)`. In the second example, its value will be `'(a b c d e f g h i)`. In the final example, its value will be `'(b d f)`.

```
(defflavor flavor-1 ((iv-a '(a b c))) ())

(defflavor flavor-2 ((iv-a '(d e f))) ())

(defflavor flavor-3 () (flavor-1 flavor-2))
```

Figure B.1: Multiple inheritance example.

```
(defherb flavor-3 ((iv-a + nil)) ())

(defherb flavor-3 ((iv-a + '(g h i))) ())

(defherb flavor-3 ((iv-a - '(a c e))) ())
```

Figure B.2: **defherb** examples.

B.3.2 Sage.Lisp

This file implements structured and abstract data type support for instance variables. Both facilities depend on storing special-purpose structures, known as **vp-slot**'s, in instance variables. Descriptions of the **vp-slot** structure, and the important functions which access it, follow (many of the concepts used here come from the Strobe system [13]):

vp-slot *[Structure]*

> A **vp-slot** contains three primary fields. The **value** field holds the "value" of the slot. The **datatype** field holds an indication of what sort of objects will reside in the **value** field of the slot. Finally, the **user-defined-facets** field holds an association list of arbitrary facet names and values; new facets may be added at any time.
>
> A **vp-slot** may be thought of as a value with arbitrary *annotations* (All slots are annotated with a **datatype** facet). These annotations might permit a program to reason about the contents of the slot when necessary.

getfacet *object slot* **&optional** *(facet 'value) errorflg novalueflg* *[Function]*

> The function **getfacet** returns the value of *facet* in *slot* of *object*. *Facet* defaults to **value**, which retrieves the **value** field of the **vp-slot**. Other acceptable bindings for *facet* are **datatype**, plus any facet in the **user-defined-facets** field of the slot. If the facet doesn't exist, and the value of *errorflg* is non-**nil**, a fatal error will occur. If the value of the facet is ***novalue***, and *novalueflg* is **nil**, the value returned from **getfacet** will be **nil**; otherwise, it will be the value found in the facet.

putfacet *object slot* **&optional** *(facet 'value) (value '****novalue****) errorflg* *[Function]*

The function **putfacet** puts *value* into *facet* of *slot* of *object*. If the facet doesn't exist, it is first created. If the slot doesn't exist (*e.g.*, the instance variable named *slot* doesn't exist, or doesn't contain an object of type **vp-slot**) and *errorflg* is non-**nil**, a fatal error is signalled.

#_ [*Reader Macro*]

Unfortunately, by placing **vp-slot** structures in instance variables of Flavor instances, it becomes impossible to simply get the "value" of the instance variable (since the value is now a **vp-slot**). The **#_** reader macro is a piece of syntactic sugar which expands to the form (**vp-slot-value** ...), and hence, retrieves the **value** field of the slot. Therefore, references to instance variables which contain slots can be preceded by **#_** to retrieve the actual value of the slot.

A number of macros are defined in terms of these basic functions; their function should be clear from examination of the source code.

Abstract Data Type Support

Abstract data type support for instance variables is implemented by forwarding messages sent to **vp-slot**'s to the objects pointed to by their **datatype** fields. Consider the example in Figure B.3. The inclusion of the **:gettable-instance-variables** option in the definition of **flavor-1** causes instances of **flavor-1** to repond to **:iv-a** messages (note the ':' in the message name); instances of **flavor-1** do *not* respond to the **iv-a** message.

Normally, when a message for which no method is defined is sent, an error occurs; however, it is possible to define an **:unclaimed-method** method for a Flavors class. The **:unclaimed-method** is invoked when an undefined message is sent. The file **sage.lisp** defines a Flavors class, **sage-class**, which has just this sort of **:unclaimed-method**.

When an undefined message is sent to a Flavors instance which has **sage-class** as an ancestor, the following steps are taken:

1. If the message is actually the name of an instance variable in the instance, the message name is evaluated (using **symeval-in-instance**) to retrieve the value of the variable.

2. If the value of the variable is a structure of type **vp-slot**, a message is sent to the Flavors instance stored in the **datatype** field of the slot. The message name is taken from the first "argument" of the unclaimed message. The arguments in the message are the Flavors instance to which the message was originally sent, the name of the instance variable to which the message was sent, and all but the first of the original arguments of the unclaimed message.

Now consider the course of events when **(send instance-1 'iv-a :get 'b)** is evaluated:

1. The message **iv-a** is received by **instance-1**.

2. **instance-1** does not handle the message **iv-a**, so the message is forwarded to the **:unclaimed-method** method defined by **sage-class**.

```
(defflavor association-list () ())

(defmethod (assocation-list :get) (instance iv key)
   (cdr (assq key (getvalue instance iv))))

(defvar assn-instance (make-instance 'association-list))

(defflavor flavor-1
   ((iv-a (make-vp-slot value '((a . 1) (b . 2) (c . 3))
                         datatype assn-instance)))
   (sage-class)
   :gettable-instance-variables)

(defvar instance-1 (make-instance 'flavor-1))
```

Figure B.3: A Flavor containing a slot

3. The **:unclaimed-method** code evaluates **iv-a** in the context of **instance-1**, and discovers the value to be a structure of type **vp-slot**. It then effectively evaluates the following: **(send assn-instance :get instance-1 'iv-a 'b)**.
4. The **:get** method of **association-list** is called. It uses its first two arguments to retrieve the association list from the **value** field of the **vp-slot** to which the message was originally directed. It then uses its third argument to return the value of an association from the list.
5. The value returned by the **:get** method of the **vp-slot**'s datatype is returned as the value of the original message.

A number of macros are defined for the convenience of programmers:

defdatatype [*Macro*]

Defines a new Flavors class suitable for use as an abstract data type. This is syntactic sugar for a combining **defflavor** and **defmethod** into one textual unit. For example, the above definition of **association-list** could have been made by evaluating:

```
(defdatatype association-list "Implements a-list dictionaries."
   (:get (instance iv key)
      (cdr (assq key (getvalue instance iv)))))
```

#$ [*Reader Macro*]

This reader macro accepts the name of a datatype class, and returns an instance of the class. If no instances of the class have been created, it creates one and stores it in a hash table (***sage-datatype-hash-table***). This reader macro is used in creating slots:

```
(defflavor flavor-1
   ((iv-a (make-vp-slot value '((a . 1) (b . 2) (c . 3))
                        datatype #$association-list)))
   ())
```

B.3.3 Datatype.Lisp and Priority-Queue.Lisp

These files use the facilities defined by **sage.lisp** and **herbs.lisp** to define a number of useful abstract data types. In general, these ADT's respond to an **:initialize** message to initialize themselves to an "empty" state, a **:put** message to add items to themselves, and a **:get** message to remove items from themselves.

queue *[Abstract Data Type]*

The queue data type implements FIFO storage in an instance variable. The current implementation uses lists maintained by the **tconc** function, defined in **datatype.lisp**. The **:initialize** message empties the queue, the **:put** message enqueues entry on the end of the queue, and the **:get** message dequeues an entry from the front of the queue.

If the instance variable in which the queue resides has a **max-length** facet, entries are added to the queue if-and-only-if the current length of the queue is less than the specified maximum length.

Two values are returned by a **:put** message. The first value is **t** if there was room to append the new entry; the second value is the value appended to the queue. Two values are also returned by the **:get** message. The first is the value found at the head of the queue; the second is **nil** if the queue was empty before the message, or **t** if it was non-empty.

All operations defined for a queue require $O(1)$ time.

dictionary *[Abstract Data Type]*

The **dictionary** is a fuller version of the **association-list** ADT described above. The **:put** and **:get** operations require $O(n)$ time, and hence, suggest the **dictionary** datatype be used when the number of entries is expected to be small. In addition to **:initialize**, **:put**, and **:get** messages, the **dictionary** also responds to the following messages:

:add *key value* *[Datatype Message]*

Adds *value* as an additional value to be associated with *key*. A **:get** message on *key* will subsequently return lists of two or more values. Requires $O(n)$ time.

:forget *key* *[Datatype Message]*

Removes the entry associated with *key* from the dictionary. Requires $O(n)$ time.

:map *function* [*Datatype Message*]

Applies *function* to each entry in the dictionary. *Function* must be a function of two arguments; the first argument will receive the key of an entry, and the second will receive the value of the key. Requires $O(n)$ time.

:new-id [*Datatype Message*]

Returns a key which is guaranteed not to be in the dictionary. This is currently implemented using **gensym**, and as such, requires $O(1)$ time.

:number-of-entries [*Datatype Message*]

Returns the number of entries in the dictionary. Requires $O(1)$ time.

:all-entries [*Datatype Message*]

Returns all of the entries in the dictionary, in association-list format. Requires $O(1)$ time.

sorted-dictionary [*Abstract Data Type*]

The **sorted-dictionary** is a variant of the dictionary which keeps its entries in sorted order, as defined by a user-supplied comparison function. It responds to the same messages as does the **dictionary**. The time complexity of operations defined for a **sorted-dictionary** are equivalent to those defined for a **dictionary**.

The comparison function must be a predicate of two arguments, and must return **t** if-and-only-if the first argument is "greater" than the second argument. For example, if the keys represent timestamps, and the dictionary is to keep the keys sorted in ascending order, the comparison function can be specified as **#'<**, the **lessp** function.

In addition to the messages defined by the **dictionary** data type, the **sorted-dictionary** also responds to these messages:

:greatest-entry [*Datatype Message*]

The **:greatest-entry** message returns the key having the "greatest" value, as defined by the comparison function. Because the dictionary is kept in sorted order, this operation requires only $O(1)$ time.

:next-entry *n* [*Datatype Message*]

The **:next-entry** message returns the key of the entry having the next "greatest" value to that of *n*. This is an $O(n)$ operation.

hash-dictionary [*Abstract Data Type*]

The `hash-dictionary` is a dictionary implementation which is based on hash tables, rather than association lists. It responds to the same messages as does the `dictionary` ADT. Its advantage over the `dictionary` is that insertion, lookup, and deletion operations are all of $O(1)$ time complexity; however, the enumeration message, `:all-entries`, is of $O(n)$ time complexity.

`monitor` *[Abstract Data Type]*

The `monitor` data type is a special purpose ADT which aids in the implementation of lexically-scoped mutual exclusion. Storage for the monitor is implemented by a `monitor` structure:

`monitor` . *[Structure]*

The monitor structure contains two fields: `owner`, which points to the `process` which currently owns the monitor; and `waiting-processes`, which is a queue of processes waiting to obtain ownership of the monitor.

`:enter` *wakeup-stream* *[Datatype Message]*

A `process` wishing to enter a region of mutual exclusion sends this message. If the monitor is unowned, the owner is set to the value of `***care-process***`, and the caller is allowed to enter the region of mutual exclusion.

If the monitor is currently owned, a dotted pair, consisting of the value of `***care-process***` and *wakeup-stream*, is enqueued on the `waiting-processes` queue of the monitor. The caller then calls `accept-packet` in order to suspend execution. When the caller's request reached the head of the queue, a packet will be sent to *wakeup-stream*, restarting the suspended caller.

`:exit` *[Datatype Message]*

The `:exit` message relinquishes ownership of the monitor, and restarts the next process waiting to obtain it (if any).

If the `waiting-processes` queue is non-empty, the first entry on the queue is dequeued. The entry contains the `process` handle of the waiting process, which is placed in the `owner` field of the monitor, and the `stream` upon which to send the wakeup message.

If the queue is empty, the `owner` field of the monitor is set to `nil`, so that the monitor is marked as unowned.

`with-monitor` *monitor-name* `&body` *forms* *[Macro]*

This macro implements an error-protected, lexically-scoped mutual exclusion. *Monitor-name* must be the name of an instance variable in the Flavors instance currently bound to `self` which holds a `monitor`. Upon entry to this macro, an `:enter` message is sent to the monitor to gain entrance. The expressions in *forms* are then executed under `unwind-protect` protection, such that if an error occurs during their execution, the monitor is guaranteed to be released.

This macro is equivalent to the `with.monitor` macro of Interlisp-D.

without-monitor *monitor-name* **&body** *forms* [*Macro*]

This macro is intended to be used within the scope of a **with-monitor** form. Its purpose is to temporarily release ownership of the monitor specified by *monitor-name* (using the **:exit** method), and then to reobtain it (using the **:enter** method) after the forms in *forms* have been executed. Typically, **forms** will contain an expression that causes the calling process to suspend for some period of time (or until a packet arrives on some stream).

This macro is similar in spirit to the **monitor.await.event** macro of Interlisp-D.

priority-queue [*Abstract Data Type*]

The **priority-queue** data type and the code needed to implement it are contained on the file **priority-queue.lisp**. The build of this file is a set of ZetaLisp routines which implement a dynamic, *Heapsort*-style priority queue. The implementation is derived from algorithms DELETEMIN and INSERT, from section 4.11 of [1]. Insertion and deletion from this queue both require $O(n \log n)$ time.

priority-queue [*Structure*]

The **priority-queue** structure implements storage at the nodes of the partially-ordered binary tree. It has fields **left-child**, **right-child**, and **item**. In addition, for convenience, it has a **priority-function** field which stores the priority-computing function for entries in the tree.

exchange-nodes ***top bottom*** [*Macro*]

This macro exchanges the contents of nodes ***top*** and ***bottom***.

insert-in-queue ***queue node*** [*Function*]

This function inserts ***node***, an instance of a **priority-queue** structure, into the tree rooted by ***queue***. It recursively descends into the tree, heading for the leftmost free node at the lowest level of the tree (creating a new level if necessary). As it unwinds from the recursion, it exchanges nodes as necessary to maintain the partial order. The value returned from this function is the new root of the tree, which may have changed.

rebalance-queue ***queue*** [*Function*]

This function rebalances the tree rooted at ***queue*** after its root has been removed.

remove-from-queue ***queue*** [*Function*]

This function removes the item from the partially-ordered tree rooted at ***queue***, and rebalances the tree to maintain the partially-ordered invariant. It returns two values: the value found at the root, and a pointer to the new root of the tree.

sorting-spec ::= (*key-spec* . *sorting-spec*) | **nil**
key-spec ::= (*key-name* . *field-spec-list*)
field-spec-list ::= (*field-spec* . *field-spec-list*) | **nil**
field-spec ::= (*field-computation* . *predicate*)
field-computation ::= *field-arg* | (*field-op* . *field-arg-list*)
field-arg-list ::= (*field-arg* . *field-arg-list*) | **nil**
field-op ::= **any-lisp-function**
key-name ::= **any-lisp-symbol**
field-arg ::= *field-number* | **'any-valued-lisp-symbol**
field-number ::= **any-lisp-integer**
predicate ::= **any-lisp-predicate**

Figure B.4: BNF Grammar for declaring sorting functions.

```
((:site ((+ (* 0 '16) 1) . <))
 (:agent (2 . alphalessp))
 (:task (3 . <)))
```

Figure B.5: A sample sorting specification.

B.4 Instrumentation for CAOS

The CARE system comes supplied with a wide variety of "instrument panels" which report how various components of the simulated execution architecture are being utilized. Much of CAOS was defined prior to the existence of these instruments, and the file **pravda.lisp** contains vestigial remnants of an interim CAOS-based instrumentation package. This package is no longer in use, and it will not be documented here, although it is part of the CAOS sources. There are, however, CAOS-specific instrument panels which are still in use. These panels are documented in this section.

B.4.1 Scrolling-Text-Panel.Lisp

The file **scrolling-text-panel.lisp** contains an instrument which displays information in a sorted order in a ZETALISP-defined window known as a **tv:scroll-window**. Such windows are designed to display a structured representation of data; new lines of information may be added or deleted dynamically, and the window may be scrolled vertically if more information is being displayed than can fit in the window.

The **scrolling-text-panel** is a **tv:scroll-window** whose sorting order and display formatting commands are specified by a simple, declarative grammar. The declaration of the sorting function is specified in the **:sorted-by** instance variable of the panel: the formatting function is specified by the **:printed-by** and **:formatted-by** instance variables. We first describe the grammar as it pertains to sorting.

The sorting grammar is described in BNF format in Figure B.4; [1] an example from CAOS appears in Figure B.5. Unquoted numbers used in *field-number* positions refer to corresponding elements of a vector in which information which drives the sorting and display functions resides.

The sorting declaration in Figure B.5 constructs three sorting functions, indexed respectively by the keywords `:site`, `:agent`, and `:task`. The `:site` sorting function is compiled into the following pieces of Lisp code:[2]

```
(defun foo-site-sorter (item-1 item-2)
   (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
         (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset))))
      (< (+ (* (nth 0 entry-1) 16) (nth 1 entry-1))
         (+ (* (nth 0 entry-2) 16) (nth 1 entry-2)))))
```

The `:agent` sorting function is a refined version of the `:site` sorting function. It expands into:

```
(defun foo-agent-sorter (item-1 item-2)
   (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
         (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset)))
         (key-2 (array-leader item-2 tv:scroll-item-leader-offset)))
      (cond ((foo-site-sorter item-1 item-2) t)
            ((equal item-1 item-2)
             (cond ((memq key-2 '(:site)) nil)
                   (t (alphalessp (nth 2 entry-1) (nth 2 entry-2))))))))
```

The `:task` sorting function is further refined, and expands to:

```
(defun foo-task-sorter (item-1 item-2)
   (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
         (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset)))
         (key-2 (array-leader item-2 tv:scroll-item-leader-offset)))
      (cond ((foo-agent-sorter item-1 item-2) t)
            ((equal item-1 item-2)
     (cond ((memq key-2 '(:site :agent)) nil)
                   (t (< (nth 3 entry-1) (nth 3 entry-2))))))))
```

We now discuss the language with which formatting functions are defined. Lines of text are output to **scrolling-text-panels** with the function **format**; in order to use this function, we must have a way of choosing both format control strings and the expressions which are evaluated to generate arguments for these control strings.

[1] In this figure, and in Figure B.6, tokens in *this font* are non-terminals, and tokens in `this font` are terminals. Occurrences of "." are Lisp "consing dots;" thus, where the grammar would ordinarily demand statements of the form `(a . (b . (c . nil)))`, it is acceptable to supply the form `(a b c)`.

[2] The arguments `item-1` and `item-2` are bound to instances of `tv:scroll-line-item` structures. The internal representation of these structures is unimportant, except that arbitrary application-program information may be stored in their *array leader* sections. The first word of available storage in the array leader is found at `tv:scroll-item-leader-offset`.

print-spec ::= (*key-spec* . *print-spec*) | `nil`
key-spec ::= (*key-name* . *field-spec-list*)
field-spec-list ::= (*field-computation* . *field-spec-list*) | `nil`
field-computation ::= *field-arg* | (*field-op* . *field-arg-list*)
field-arg-list ::= (*field-arg* . *field-arg-list*) | `nil`
field-op ::= `any-lisp-function`
key-name ::= `any-lisp-symbol`
field-arg ::= *field-number* | `'any-valued-lisp-symbol`
field-number ::= `any-lisp-integer`

Figure B.6: BNF Grammar for declaring printing functions.

```
((:site . "SITE-~D-‘D")
 (:agent . "   ~A ~A (~D run, ~D wait)")
 (:task . "       ~A ~A ~A"))

((:site 0 1)
 (:agent 2 (car 3) 4 5)
 (:task 4 3 5))
```

Figure B.7: A sample formatting specification

Format control strings are chosen by indexing into an association list stored in the `formatted-by` instance variable of the panel. Lisp expressions which generate the arguments for `format` are created by parsing expressions defined by the grammar in Figure B.6 and are found in the `printed-by` instance variable of the panel. The contents of these two instance variables, in an example from the CAOS instrumentation, is illustrated by Figure B.7. The panel defined by the specifications in Figures B.5 and B.7 will display sites in column-major order; within each site, agents will be displayed alphabetized by name; within each agent, tasks will be displayed ordered by arrival time. For example:

```
SITE-1-1
   MERGESORT-MANAGER-1 INITIALIZED (0 run, 0 wait)
   MERGESORTER-1-1 INITIALIZED (1 run, 3 wait)
      RUNNING 345700 NEIGHBOR
      NEVER-RUN 345792 MERGESORT
SITE-1-2
   MERGESORTER-1-2 INITIALIZED (0 run, 0 wait)
```

B.5 CAOS Structures and Macros

The file `czardefns.lisp` contains macro and structure definitions for the rest of the CAOS system.

request-message *[Structure]*

The **request-message** structure is a list which defines the contents of messages sent using the various *post* operators of CAOS.

response-message *[Structure]*

The **response-message** structure is a list which defines the contents of messages sent as responses to value-desired messages.

caos-time *[Macro]*

This macro retrieves the current simulator time, which is measured in simulator clock units. Presently, this figure is measured in 10 nanosecond units.

runnable-item *[Structure]*

The **runnable-item** is the CAOS scheduler's handle on a process. Most of its structure was described in Section 5.4. The **panel-entry** field holds the **tv:scroll-window** line entry of the process.

contract *[Resource]*

Resources are Lisp objects which must be explicitly allocated and deallocated. This is counter to the normal Lisp philosophy, but is quite useful when the extent of an object is known. The advantage of declaring objects as resources is that large numbers of unused copies of the objects aren't accumulated to be reclaimed only when the garbage collector is run. The **contract** resource allocates and deallocates **runnable-item**'s.

care-site-scrolling-panel-entry *[Structure]*

This structure is the vector which holds information for sorting and formatting **care-site** entries in the **scrolling-text-panel**. In figures B.5 and B.7, this structure is referenced by printing and sorting specifications keyed by **:site**. The fields of the structure are:

x, y: Coordinates of the site in the processor grid.

state: The condition of the site.

agent-scrolling-text-panel-entry *[Structure]*

This structure is the vector which holds information for sorting and formatting agent entries in the **scrolling-text-panel**. It is referenced by printing and sorting specifications keyed by **:agent**. The fields of this structure are:

x,y: Coordinates of the site upon which the agent is located.

name: The name of the agent.

state: The condition of the agent.

nrun: The number of runnable tasks in the agent.

nwait: The number of suspended tasks in the agent.

task-scrolling-panel-entry *[Structure]*

This structure is the vector which holds the information for sorting and formatting task (process) entries in the **scrolling-text-panel**. This structure is referenced by printing and sorting specifications keyed by **:task**. The fields of the structure are as follows:

x, y: Coordinates of the site upon which the task is executing.

name: The name of the agent in which the task is executing.

entry-time: The simulator time at which the task started.

state: The current state of the task.

message: The name of the message being executed by the task.

future *[Structure]*

A **future** is a special object which represents a promise of a value to be returned by a remote computation. It has the following fields:

value: When the future has a value, it is placed in this field.

msg-id: The unique id of the message which associated with the computation which will return a value to this future.

waiting-processes: The number of processes waiting for the future to have a value.

waiting-process-list: The list of processes waiting for the future, in **tconc** format.

single-assignment: A boolean field; true if the future can only be assigned a value once.

original-message: The contents of the **request-message** message sent to start the remote computation which will return a value to this future. Used when a clocked, single-assignment future is reposted.

destinations: The destination agents to which the original message was sent; used by **repost**.

multi-future *[Structure]*

A **multi-future** is a collection of futures. It is returned by the value-desired, multipost-style messages. A **multi-future** contains a lists of satisfied and unsatisfied futures. Initially, all futures in a **multi-future** are unsatisfied; as values of remote computations are received, unsatisfied futures are given values and moved to the list of satisfied futures.

B.6 Declaring CAOS Agents

The file **czardecl.lisp** contains routines to declare sites and agents.

defsite *[Macro]*

This macro makes it possible to declare Flavor classes which implement site-global storage within CAOS. **defsite** is defined in terms of **defherb**, and thus, it is possible to define instance variables within site instances which support abstract data type operations.

It is conceivable that if CAOS were ever implemented on a heterogeneous array of processors, there would be a number of site types, perhaps defined in a taxonomy.

vanilla-site *[Site]*

Instances of **vanilla-site** implement site global storage. Each instance has the following instance variables:

static-agent-stream-table: Contains a dictionary which maps static (named) agents to their input stream addresses.

unresolved-agent-stream-table: Contains a dictionary which maps the names of remote agents not yet known during initialization to the addresses of streams in local agents which have requested the addresses of the unknown remote agent.

local-agents: A dictionary which maps the names of local agents to their addresses.

free-process-queue: A queue which holds information allowing free processes to be reused in preference to creating new processes.

care-site: Holds a pointer to the CARE site structure for the site upon which the **care-site** is located.

locale: Holds a CARE-defined structure which is created by **make-locale**, and which is updated by **update-locale**. Each call to **update-locale** modifies the structure so that a call to **locale-site** returns the least-recently-referenced site in the locale. This is a simple approach to load-balancing.

incoming-stream: Holds the stream upon which the site manager listens for site-oriented requests.

defagent-keyword *[Macro]*

This macro defines the syntax for a new keyword used in a call to **defagent** (see below). The keywords described in Chapter 4, plus a number of keywords not described, are all declared through the use of **defagent-keyword**.

defagent *[Macro]*

The **defagent** macro, which is defined in terms of **defherb**, is the basic form by which new agents are declared. It is described in detail in Chapter 4.

defagent-method *[Macro]*

The **defagent-method** macro is syntactic sugar for **defmethod**, but has the advantage of being able to define the same method for multiple message names.

clock *[Abstract Data Type]*

The **clock** ADT responds to the **:rearm**, **:tick**, and **:stop** messages. The **value** field of a **vp-slot** of the **clock** datatype holds a list of messages to be executed when the clock "fires."

vanilla-agent *[Agent]*

The **vanilla-agent** is the most basic agent in the system. It has the following instance variables:

local-process-stream-table: A dictionary which maps from a process handle to a utility **stream** the process uses to wait for wakeup messages.

outstanding-message-table: A dictionary which maps from ids of messages to their associated futures.

runnable-process-list: A priority queue which implements the scheduling policy defined for the agent.

scheduler-lock: A monitor data type which is used to implement mutual exclusion around routines which modify the agent scheduler database.

process-table: A dictionary which maps from CARE process handles to CAOS **runnable-items**.

self-address: The stream upon which the agent's input process listens for requests and responses from other agents.

priority-queue-context: Holds information for creating nodes in the **runnable-process-list** priority-queue.

care-site: Points to the **care-site** structure for the site upon which the agent is located.

symbolic-name: Holds the name of the agent. Statically-created agents are named by the application program; dynamically-created agents are named by CAOS, using **gensym**.

agent-scheduler: Holds the CARE process handle of the process which is currently performing the duties of the agent scheduler.

running-processes: Holds a list of **runnable-item**'s which represent processes handed off to CARE for execution.

symbolically-referenced-agents: Holds a list of other agents to be referenced by name by methods executing within the context of the agent.

`initial-forms`: A list of expressions to be evaluated after CAOS has been initialized. The purpose of these forms is to initialize an application.

`:select-process-fifo` *item-1 item-2* [*Method of* `vanilla-agent`]

This method implements FIFO scheduling of tasks within an agent. It is called as the priority function for the `priority-queue` stored in the `runnable-process-list`. Priorities are derived by comparing the `time-stamp` fields of *item-1* and *item-2*, which are `runnable-item`'s.

`process-agenda-agent` [*Agent*]

The `process-agenda-agent` is a subclass of `vanilla-agent`. It differs from `vanilla-agent` in that certain message names may be given execution priorities. Such priorities are defined by specifying message names in order in a list stored in the `process-agenda` instance variable; messages at the front of the list have higher priority than those at the end of the list.

`:select-process-agenda-timestamp` *item-1 item-2* [*Method of* `process-agenda-agent`]

This method implements "agenda-based" scheduling of tasks in an agent. It is the priority function for the `runnable-process-list`. Priorities are derived by first comparing the `message-name` fields of *item-1* and *item-2*; if these fields are the same, the function then compares the `time-stamp` fields, as in the FIFO scheduler above.

B.7 Initializing a CAOS Application

The file `czarinit.lisp` contains the code which initializes CAOS at the start of a run. Initialization occurs in two distinct phases: one, *static*, before the CARE simulator is started, and the other, *dynamic*, just after.

The first set of functions, macros, and methods in `czarinit.lisp` is involved in static initialization. During this phase, the application initialization file (see Figure 4.4 and Appendix A) is read and interpreted. As a result of interpreting this file, all statically-declared agents are created on the appropriate sites, and the messages which initialize the application once CAOS is running are stored away.

`:init` [`:after` *Method of* `care-site`]

During the static phase, new instances of `care-site` Flavor instances are created. The `:init` method is primarily responsible for initializing all of the abstract data types which are part of the `care-site`.

`:init` [`:after` *Method of* `vanilla-agent`]

When a new agent instance is created, the `:init` method initializes a number of abstract data types, and also adds an entry to the appropriate `care-site`'s `local-agents` dictionary.

make-initial-agent *agent-class global-name care-site* [*Macro*]

This macro is invoked when the **caos-initialize** form is interpreted. *Agent-class* is the name of an agent class as defined by **defagent**. *Global-name* is the name by which this instance of the agent class will be known throughout the processing grid. *Care-site* is a two-element list specifying the x and y coordinates of the **care-site** upon which the new agent will be created. When the macro is executed, an instnace of *agent-class* with name *global-name* is created on *care-site*.

initial-agent-record [*Structure*]

This structure defines the a three-tuple with fields **name**, **class**, and **location**. Instances of this tuple make up the *agent-instances* argument to the **caos-initialize** macro (below). The **initial-agent-record** also defines the argument list to **make-initial-agent**.

caos-initialize *agent-instances initial-messages* [*Macro*]

Calls to this macro are the means by which CAOS applications are initialized. *Agent-instances* is a list of **initial-agent-record** structures. *Initial-messages* is a list of expressions to be evaluated when CAOS has finished initializing.

When a **caos-initialize** form is evaluated, four major activities occur.

1. All statically-declared agents are created by mapping over *agent-instances* and calling **make-initial-agent** on each element.
2. An agent of class **initial-agent** is defined. The **initial-agent** class is a subclass of **vanilla-agent** which makes reference to all other statically-declared agents.
3. An instance of the **initial-agent** class, called 007 is created on site (1, 1).
4. The *initial-messages* argument is used to define an **:initial-form** method for the class **initial-agent**.

The remainder of **czarinit.lisp** is devoted to dynamic initialization. The necessary site and agent instances were created during the static phase; during the dynamic phase, these structures must be linked up with CARE. Dynamic initialization consists of starting the site manager processes in each of the sites, starting the input monitor and scheduler processes in each of the agents, and exchanging the names and addresses of each of the agents in order to resolve symbolic references. Dynamic initialization is completing by sending agent 007 an **:initial-form** message.

start-czar *initializer-stream* [*Process*]

The **start-czar** process is the first process run once CARE starts. It drives all dynamic initialization tasks, as follows:

1. Creates a site manager process in each site.
2. Waits for each site manager process to return the address upon which it listens for requests.

3. Creates a process on each site that contains a statically-declared agent, whose task is to initialize those agents.
4. Waits for each site containing statically-declared agents to indicate its agents are initialized.
5. Sends the `:initial-form` message to the agent named 007.

`start-site` *initializer-stream site-stream* *[Process]*

This process is the CAOS site manager. Upon start-up, it sends the value of *site-stream* to *initializer-stream* (upon which the `start-czar` process is waiting). It then enters an endless loop in which it responds to service requests directed to *site-stream*. The specific services implemented by the site manager were discussed in Section 5.2.

`start-agents` *all-care-sites-list start-agents-stream* *[Process]*

This process is responsible for initializing statically-declared agents on each site. For each agent, it does the following:

1. Starts the input monitor process.
2. Broadcasts a `:new-initial-agent-online` message, containing the agent's name and the address upon which its input monitor process listens, to all other site managers in the grid (the value of *all-care-sites-list*).
3. For each agent named in the agent's `symbolically-referenced-agents` instance variable, sends a `:request-symbolic-reference` message to the site manager, and waits for a response.
4. Sends a message to the `start-czar` process indicating that the site is ready to run.

B.8 The CAOS Runtime System

The file `czar.lisp` contains the "runtime" system for CAOS. The functions documented in sections 4.3 and 4.4 are implemented by in this file. In what follows, we document those functions upon which the functions in these sections depend.

`agendize` *future* *[Defun-Method of* `vanilla-agent`*]*

This is the low-level function used to suspend a process until *future* receives a value. It sets the calling process's state to `:suspended`, adds the process's `runnable-item` to the list of processes waiting for *future*, sets the `context` field of the `runnable-item` to be the process's wakeup stream, and sends to itself the `:reschedule` message, which invokes the scheduler to put the process to sleep. Upon waking up, it sets the process's state to `:running`, and returns to its caller (typically, `value-future`).

`multi-agendize` *multi-future* *[Defun-Method of* `vanilla-agent`*]*

This function is the multi-future version of **agendize**.

remote-address-enumerating-functions [*Variable*]

This variable holds an association list which maps ZETALISP data types into a function, which when applied to an object of the associated type, returns a list of remote addresses. This allows application programs built on top of CAOS to represent collections of agents in forms other than lists.

coerce-destination *dest-stream* [*Defun-Method of* **vanilla-agent**]

This function coerces *dest-stream*, which may be a remote address, a future, or the name of an instance-variable in **self** into a stream.

If *dest-stream* is a **remote-address**, it is returned unmodified. If *dest-stream* is a symbol, it is evaluated in the context of **self**, and is expected to evaluate to a **remote-address** (this is the mechanism by which application programs are able to refer to statically-declared agents by name). Finally, if *dest-stream* is a **future**, **coerce-destination** calls **value-future** to retrieve the destination **remote-address**.

list-of-remote-addresses *list* [*Defun-Method of* **vanilla-agent**]

This is the enumerating function for lists of remote addresses.

enumerate-destinations *remote-addresses* [*Defun-Method of* **vanilla-agent**]

This function uses ***remote-address-enumerating-functions*** to coerce *remote-addresses* into a list of **remote-address**'s.

stream-send *dest-stream priority flags message args* [*Defun-Method of* **vanilla-agent**]

This function is a common subfunction used by CAOS-defined posting operators. It uses the facilities of CARE to send *message* and *args* to *dest-stream* with CARE priority *priority*. *Flags* is a list which controls the operation of **stream-send**. The following symbols may be included in *flags*:

:no-return —Causes **stream-send** to send a side-effect message.

:return-future —Causes **stream-send** to create a **future**, assign it a unique identifier, send the message with **self-address** as the return address, and return the new future to the caller.

:return-multi-future —Like **:return-future**, but causes **stream-send** to create and return a **multi-future** instead of a **future**.

:single-assignment —Causes **stream-send** to create a *single-assignment* **future**, a **future** whose value can only be set once.

make-and-initialize-future *type* [*Defun-Method of* **vanilla-agent**]

This function creates a new future of type *type* (either **future** or **multi-future**). It also generates a unique identifier for the future in the agent's **outstanding-message-table**, and places the future in the table, keyed by the unique identifier.

format-stream-request *id stream message args* [*Function*]

This function formats a message and its arguments for transmission to another agent. *Id* is the unique id of the message; *stream* is the stream to which answers should be directed.

agent-input-process *agent request-stream* [*Process*]

This process is the process which monitors **self-address** for requests and responses from other CAOS agents. It is created exactly once per agent, and performs the following initialization steps:

1. Sets **self-address** to the value of *request-stream*.
2. Creates the agent scheduler process.
3. Arms all clocks in the agent.

After initializing the agent, **agent-input-process** enters a loop, in which it waits for messages directed to **self-address**, and then processes them accordingly.

:handle-request *request for-effect* [*Method of* **vanilla-agent**]

This method is invoked by the input monitor process when a request message is received. It allocates a new **runnable-item**, and fills in its fields by copying from *request*, a **request-message** structure.

It then sends the new **runnable-item** to the scheduler process. If the scheduler is idle when this method is invoked, the **runnable-item** is sent to the process in a CARE message (this reawakens the idle scheduler); otherwise, the **runnable-item** is simply enqueued on the agent's **runnable-process-list**.

:handle-response *response* [*Method of* **vanilla-agent**]

This method is invoked when the input monitor process encounters a **reponse-message**. It first checks if the response is directed towards a **future** or a **multi-future**. In the latter case, it calls upon the **:handle-multi-reponse** method to process the response. In the former case, it does the following:

1. If the future associated with the response is a single-assignment future, the future is removed from the agent's **outstanding-message-table**.
2. The value is removed from the response, and placed in the **value** field of the future.
3. The **satisfied** field of the future is set to **t**.
4. The **:run-processes** method is invoked, which restarts all processes waiting on the future.

`:handle-multi-reponse` *multi-future value source* [*Method of* `vanilla-agent`]

This method is called when a response to a `multi-future` is received. *Source* is a cons of the sending agent's name and self-address; individual `future`'s in the `multi-future` may be keyed by either.

The method uses *source* to find the appropriate `future` in the `multi-future`'s `unsatisfied-future` list, and places *value* in its `value` field. If the `multi-future` is in `:any` wakeup mode, all processes waiting on the future are reawakened; if the `multi-future` is in `:all` mode, the waiting processes are reawakened only if there are no more unsatisfied `future`'s.

`agent-scheduler` *agent scheduler-process-stream* [*Process*]

This process is the CAOS scheduler process for agents. It is written as a loop which performs the following operations:

1. If the scheduler has previously determined that there are no runnable processes, or if there are requests waiting in the `runnable-process-stream`, the scheduler tries to get the next request from the `runnable-process-stream`. If neither condition is true, the scheduler skips to step 3, below.
2. If the message is a symbol, it is the name of a clock which has just ticked; in this case, the scheduler sends the `:tick` message to the clock.

   If the message is a `runnable-item`, it is a request to the scheduler to perform an operation on the associated process. To be sent to the scheduler, the state of the process must be either `:suspended` or `:never-run`. In either case, the scheduler adds the item to the `runnable-process-list`.
3. The scheduler next tries to hand to CARE for execution as many processes as it can. The number of processes it is allowed to run at any one time is determined by the value of `*number-of-running-agent-processes*`.
4. Finally, the scheduler checks to see if any special conditions are outstanding. One special condition is that the user has requested a breakpoint (*e.g.*, to perform some debugging with the CARE clock shut off). The other special condition is that it is about to be too late to perform an immediate garbage collection; in this case, the scheduler shuts off the CARE clock, and calls `gc-immediately`, the ZETALISP function which invokes the garbage collector.

`:add-to-runnable-process-list` *item* [*Method of* `vanilla-agent`]

This method enqueues a `runnable-item` on the agent's `runnable-process-list`. If the CAOS instrumentation package is enabled, it also adds a line representing the process to the `scrolling-text-panel`.

`:choose-next-runnable-item` [*Method of* `vanilla-agent`]

This method removes the highest-priority `runnable-item` from the `runnable-process-list`, unless the number of processes already handed to CARE is greater than or equal to `*number-of-agent-running-processes*`.

If the CAOS instrumentation package is enabled, and an item was removed from the queue, this method also removes the line representing the process from the `scrolling-text-panel`.

`:schedule-next-process` *return-new-items* [*Method of* `vanilla-agent`]

This method is called by the scheduler process to hand the highest-priority process to CARE for execution. If the state of the process is `:never-run`, the `:create-new-process` method is invoked to create a new process. If the state of the process is `:runnable`, the process is reawakened by calling the function `resume-old-item`.

`:reschedule` *future* [*Method of* `vanilla-agent`]

This method is invoked to suspend a process until *future* has a value. It first updates the CAOS instrumentation, then tries to run as many processes as possible (to keep the processor as busy as possible), and then suspends, waiting for a packet on its wakeup stream. Upon reawakening, it updates the CAOS instrumentation once again, and returns to its caller (typically `agendize`).

`:create-new-process` *runnable-item* [*Method of* `vanilla-agent`]

This method is called to create a new application-level process. It preferentially recycles a process waiting in the `free-process-queue` of the `care-site` associated with the agent. If there are no free processes available, it creates a new process using the facilities of CARE.

`message-handler` *agent runnable-item wakeup-stream* [*Process*]

All CAOSpostings are executing in processes in which `message-handler` is the top-level. This process is a loop, which does the following:

1. Executes the message and arguments contained in *runnable-item*, an instance of a `runnable-item`.
2. Tries to pull the next `runnable-item` in state `:never-run` off the `runnable-process-list`. If there is such an item, `message-handler` returns to step 1 with *runnable-item* set to the new `runnable-item`.
3. Otherwise, the process queues itself on the `free-process-queue` of its associated `care-site`, to be reused later. It does this by calling the function `wait-for-an-item`.

`czar-initialize` *dimensions file aux-display* [*Function*]

This function is called to start CAOS. It initialize a number of global variables, sets up the CAOS instrumentation, and reads the *file*, the application file which contains the `caos-initialize` form.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structure and Algorithms*. Addison-Wesley, 1983.

[2] N. C. Aiello, C. Bock, H. P. Nii, and W. C. White. *Joy of AGE-ing*. Technical Report, Heuristic Programming Project, Stanford University, 1981.

[3] H. Brown, C. Tong, and G. Foyster. PALLADIO: An Exploratory Environment for Circuit Design. *IEEE Computer*, 16, December 1983.

[4] H. I. Cannon. *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*. Technical Report, A.I. Lab, Massachusetts Institute of Technology, 1981.

[5] B. A. Delagi. *The* CARE *User Manual*. Technical Report, Knowledge Systems Laboratory, Stanford University, 1986. In preparation.

[6] Denelcor, Inc. *Heterogeneous Element Processor: Principles of Operation*. February 1981.

[7] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*, 12:213–253, June 1980.

[8] R. P. Gabriel and J. McCarthy. Queue-Based Multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[9] R. H. Halstead, Jr. Implementation of MultiLisp: Lisp on a Multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[10] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[11] V. R. Lesser and D. D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *The AI Magazine*, 15–33, Fall 1983.

[12] E. Y. Shapiro. *Lecture Notes on the Bagel: A Systolic Concurrent Prolog Machine*. Technical Memorandum TM-0031, Institute for New Generation Computer Technology, November 1983.

[13] R. G. Smith. *Structured Object Programming in Strobe.* Technical Report SYS-84-08, Schlumberger-Doll Research, March 1984.

[14] R. G. Smith and P. Friedland. *Unit Package User's Guide.* Technical Report HPP-80-28, Heuristic Programming Project, Stanford University, December 1980.

END

12-86

DTIC